UNISYS

A16 System Architecture MCP/AS (Extended)

Support Reference Manual

Copyright © 1993 Unisys Corporation
All rights reserved.
Unisys is a registered trademark of Unisys Corporation.

January 1993

Printed in U S America
Priced Item
3950 8825-100

NO WARRANTIES OF ANY NATURE ARE EXTENDED BY THIS DOCUMENT. Any product and related material disclosed herein are only furnished pursuant and subject to the terms and conditions of a duly executed Program Product License or Agreement to purchase or lease equipment. The only warranties made by Unisys, if any, with respect to the products described in this document are set forth in such License or Agreement. Unisys cannot accept financial or other responsibility that may be the result of your use of the information in this document or software material, including direct, indirect, special, or consequential damages.

You should be very careful to ensure that the use of this information and/or software material complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

RESTRICTED RIGHTS LEGEND. Use, reproduction, or disclosure is subject to the restrictions set forth in DFARS 252.277-7013 and FAR 52.227-14 for commercial computer software.

Correspondence regarding this publication should be forwarded to Unisys Corporation either by using the Business Reply Mail form at the back of this manual or by addressing remarks directly to Unisys Corporation, East Coast Development Center, SPG Product Information, P.O. Box 203, Paoli, PA, 19301-9898, U.S.A.

UNISYS

Product Information Announcement

MAR 01 1995

o New Release • Revision o Update o New Mail Code

Title

A16 System Architecture MCP/AS (Extended) Support Reference Manual

This Product Information Announcement announces the release of Revision 1 to the A16 System Architecture MCP/AS (Extended) Support Reference Manual.

This reference manual describes the system concepts needed as background to understand the hardware/software interface of A16 systems running under MCP/AS (Extended).

Revision 1 provides added and updated information to further clarify the A16 system concepts.

To order this manual, order 3950 8825-100.

To order additional copies of this document:

- United States customers should call Unisys Direct at 1-800-448-1424.
- All other customers should contact their Unisys Subsidiary Librarian.
- Unisys personnel should use the Electronic Literature Ordering (ELO) system.

Announcement only:	Announcement and attachments:	System: A16
ASO1	ECCA	Release: Date: January 1993 Part Number: 3950 8825-100

Page Status

Page	Issue
iii	-100
iv	Blank
v through vii	-100
viii	Blank
ix through xv	-100
xvi	Blank
xvii	-100
xviii	Blank
1-1 through 1-14	-100
2-1 through 2-21	-100
2-22	Blank
3-1 through 3-31	-100
3-32	Blank
4-1 through 4-137	-100
4-138	Blank
A-1 through A-17	-100
A-18	Blank
B-1 through B-3	-100
B-4	Blank
C-1 through C-6	-100
D-1 through D-15	-100
D-16	Blank
E-1 through E-11	-100
E-12	Blank
F-1 through F-2	-100
G-1 through G-56	-100
H-1 through H-15	-100
H-16	Blank
Index-1 through Index-18	-100

Unisys has implemented a document numbering system in which the suffix of the document number (1234 5678-xyz) allows you to determine the relationship between the base document and revisions, updates, and erratas. The first digit of the suffix (x) designates revision levels; the second digit (y) designates updates; and the third digit (z) designates erratas.

About This Reference Manual

Purpose

This manual describes the system concepts needed as background to understand the hardware/software interface of A16 systems running under MCP/AS (Extended).

Scope

Incorporated in this manual are descriptions of supported data types, the processor state, and operator set. In addition there are appendices that list the alphabetical operator codes, the numeric operator codes, phased out operators, common actions, interrupts, and the modes for operator encoding.

Audience

The primary audience for this manual are systems analysts, whose responsibilities include the need to gather and analyze information in order to provide software support for an A16 running under MCP/AS (Extended).

Prerequisites

Users of this manual are expected to have full knowledge of the capabilities, features, and operation of an A16.

How to Use This Document

This manual is intended to be used as a reference document. As such, it is expected that the reader will use it to research information that is currently needed, rather than read it cover to cover. With this use in mind, the manual includes a comprehensive table of contents and is thoroughly indexed to help you find information quickly.

Organization

Section 1. Process Model

This section discusses the concepts associated with the process model that is implemented by the A16 processor. It introduces fundamental data types and the processor state required for functional specification of the operator set.

Section 2. Data Structures and Types

This section introduces most data types that can be manipulated by user program operators, as well as additional data types that are visible only to primitive programs.

Section 3. Interrupts

This section describes the interrupts and interrupt mechanism that are valid on A16 systems.

Section 4. Operator Set

This section specifies the operator formats and code streams that are used on A16 systems.

Appendix A. Alphabetical Operator Code List

This appendix provides a quick reference table that is an alphabetical list of all the operator codes, their appropriate mnemonics, encoding, type, and applicable parameters.

Appendix B. Boolean Accumulator X-Ref

This appendix is a list of operators that interact with the four Boolean Accumulators and indication of the type of interaction.

Appendix C. Operator Encoding

This appendix provides encoding charts for the primary, variant, and edit operators.

Appendix D. Data Type Formats

This appendix illustrates data word formats.

Appendix E. Restricted Operators

This appendix describes the operators that are restricted to low-level code and are unavailable in user programs.

Appendix F. Operators To Be Phased Out

This appendix describes the operators that are only available in earlier A16 systems.

Appendix G. Operator and Common Action Reference

This appendix lists each operator and common action, in alphabetical order by mnemonic.

Appendix H. Interrupt Reference

This appendix lists each interrupt in alphabetical order of the interrupt name.

An index follows Appendix H.

Results

By using this manual, a systems analyst should be able to understand the hardware/software interface of an A16 running under MCP/AS (Extended), including the operator set. This background is essential for system support, particularly in the analysis of problems that arise in system software and some user programs.

Contents

About This Reference Manual v

Section 1. Process Model

Addressing Granularity: Words 1-1
Segments and Descriptors 1-1
Actual Segments and Actual Segment Descriptors (ASDs) 1-2
Virtual Segments and Virtual Segment Descriptors (DDs and CSDs) 1-2
Procedures: Activation Records 1-3
Processes: Stacks 1-4
Programs: Code Segment Dictionaries 1-5
Linkages and Addressing Environments 1-5
References 1-7
Protection 1-8
Accessing Segments 1-8
Access Control 1-9
Addressing 1-10
Procedure Entry/Exit 1-10
Expression Stack and Stack Frames 1-11
Stack and Display Registers 1-12
Executable Code Streams 1-13
General Boolean Accumulators 1-13
ASD Memory Architecture 1-14

Section 2. Data Structures and Types

Data Words 2-2
Operands (Single and Double Precision) . . . 2-2
Real Numeric Operands 2-3
Decimal Sequence Operands 2-4
Boolean Operands 2-4
Tag-4 Word and Tag-6 Word 2-5
Program Code Words 2-5
Actual Segment Descriptors (ASD) 2-5
Actual Segment Descriptor, First Word (ASD1) . . . 2-6
Actual Segment Descriptor, Second Word (ASD2) . 2-6
Absolute Store Reference Word (ASRW) 2-7
Virtual Segment Descriptor (VSD) 2-7
Untouched DD 2-7
Unindexed (Original and Copy) DD 2-8
IndexedDD (Indexed Data Descriptor) 2-9
Code Segment Descriptor (CSD) 2-10
Virtual Segments 2-10
Virtual Index 2-11
References 2-12
Address Couples 2-12
Environment Links 2-13
Stuffed Indirect Reference Word (SIRW) . . . 2-14
Program Control Word (PCW) 2-15
Stack Structure and Linkage 2-16
Stack Frame Linkage 2-16
Top-of-Stack Linkage 2-18
Protected Object Word (POW) 2-18
Tag Values 2-19
Data Structure Access 2-20

Section 3. Interrupts

Interrupt Mechanism 3-1
Superhalt 3-1
Vectored Interrupt Conditions 3-2
Interrupt Formats 3-2
Code-Stream Designation and Resumption . . . 3-3
Interrupt Parameters 3-4
Restart Configuration 3-5
Operator Subdivision 3-5

Interrupt Definition 3-6
Arithmetic Interrupts 3-7
Service Interrupts 3-10
Enforcement Interrupts 3-16
Alarm Interrupts 3-25
External Interrupts 3-28
Interrupt Vector 3-29

Section 4. Operator Set

General Information 4-1
Operators and Code Streams 4-1
Primary, Variant, and Edit Operators 4-2
Common Actions 4-2
Order of Execution 4-2
Interrupts and Resumption 4-3
Preconditions and Invariants 4-3
Restart Mode 4-3
Operator Applicability 4-4
Expression Stack Control 4-5
Top-of-Stack Push Operations 4-6
Top-of-Stack Pop Operations 4-6
Stack-State Transformation 4-7
Descriptor Interpretation 4-8
Computational Operators 4-8
Numeric Operand Interpretation 4-8
Representable Values 4-9
Arithmetic Exceptions 4-11
Rounding Function R 4-13
Magnitude Round-to-Integer Function Ri 4-15
Algebraic Round-to-Integer Function Rai 4-16
Truncation Function T 4-17
Magnitude Truncate-to-Integer Function Ti . . 4-18
Normalization Function N 4-19
Approximation Functions 4-20
Numeric-Interpretation Operators 4-21
Arithmetic Operators 4-21
Numeric Relational Operators 4-27
Range Test Operators 4-27
Numeric Type Transfer Operators 4-28
Scale Left 4-31
Scale Right 4-32
Binary To Decimal Conversion 4-36
Mathematical Function Operator 4-38

Bit Vector Interpretation 4-38
Logical Operators 4-38
Logical Relational Operator 4-39
Literal Operators 4-40
Bit-Vector Type-Transfer Operators 4-41
Evaluate Word Structure Operators 4-43
Word Manipulation Operators 4-44
Linear Index-Function Operator 4-49
Reference Generation and Evaluation Operators 4-50
Double Precision 4-50
Stack References 4-50
Environment Link Evaluation 4-50
Evaluation of References 4-51
Address Couple Evaluation 4-51
SIRW Evaluation 4-51
IndexedWordDD Evaluation 4-52
PCW Evaluation 4-52
Reference Chains 4-52
Reference Generation Operators 4-54
Reference Modification Operators 4-60
Descriptor Attribute Extraction Operators . . . 4-62
Read Evaluation Operators 4-63
Normal Store Operators 4-70
Overwrite Operators 4-72
Interlock Operations 4-74
Processor State Operators 4-79
Code Stream Pointer Distribution 4-79
Branching Operators 4-79
Static Branches 4-79
Dynamic Branches 4-80
Stack Frame Operators 4-81
Procedure Entry Operators 4-82
Procedure Exit Operators 4-85
Assertion Operators 4-88
Top-Of-Stack Operators 4-88
Processor-State Manipulation Operators . . . 4-90
Data Array Operators 4-92
Paged Array Operations 4-92
Searching Operators 4-93

Pointer Operators 4-95
Unconditional Character-Transfer Operators . 4-100
Character Relational Operators 4-101
Scan Operators 4-101
Transfer Operators 4-102
Character-Sequence Compare Operators . . . 4-103
Character Set Membership Operators 4-105
Scan Operators 4-106
Transfer Operators 4-106
Character Translate Operator 4-107
Character Sequence Extraction Operator . . . 4-108
Decimal Character Sequence Operators 4-109
Pack Operators 4-110
Unpack Operators 4-111
Input Convert Operators 4-113
Word Transfer Operators 4-114
Edit Operators 4-115
Enter-Edit Operators 4-115
Edit Mode Operators 4-119
Miscellaneous Edit Operators 4-124
Pointer Index Restrictions 4-124
Vector Operators 4-125
Vector Assignment Operators 4-127
Dyadic Vector Operators 4-127
Dyadic Vector Operators with Scalar . . . 4-129
Scatter/Gather Operators 4-130
Reduction Operators 4-131
Linear Sum 4-131
Quadratic Sum 4-132
Recurrence Operators 4-132
Search Operators 4-134
Miscellaneous Operators 4-135
Program Restrictions Due to Hidden State 4-136

Appendix A. Alphabetical Operator Code List

Appendix B. Boolean Accumulator X-Ref

TFFF (true/false flip flop) B-1
OFFF (overflow flip flop) B-2
EXTF (external sign flip flop) B-2
FLTF (float flip flop) B-2
TFFF, OFFF, and EXTF B-3
FLTF B-3

Appendix C. Operator Encoding

Primary Mode Operators C-2
Variant Mode Operators C-4
Edit Mode Operators C-6

Appendix D. Data Type Formats

Appendix E. Restricted Operators

Program Restrictions Due to Hidden State E-10

Appendix F. Operators Being Phased Out

Appendix G. Operator and Common Action Reference

Appendix H. Interrupt Reference

Figures

2-1. Word Format 2-1

Tables

A-1. Operators, Alphabetical Listing A-1

Section 1
Process Model

This section discusses concepts associated with the process model that is implemented by the A16, as well as all other MCP/AS (Extended) systems. It introduces fundamental data types and the processor state required for functional specification of the operator set.

Addressing Granularity: Words

The unit of addressing in the A Series systems is the *word*, a memory unit made up of 48 data bits and a tag. Operators exist that can deal with parts of words; however, memory is addressed and accessed in whole words. The term *item* is more general than *word*; an item contains one or two words.

The tag determines, at least in part, the item type. Tags on MCP/AS (Extended) systems are four bits wide.

The most common data type is the single-precision operand with a tag of zero. Such words are used for numbers (integer and floating point), Boolean values, characters, and arbitrary encodings. For some purposes, pairs of words are joined to form double precision operands with tags equal to two. Most other tag values are used for words with special functions that are recognized by the processor. Some of these data types are distinguished entirely by their tag, some by a combination of tag and other fields within the word, and some by a combination of tag and context.

Segments and Descriptors

An aggregate of memory words associated for addressing purposes is called a segment. Segments may contain either data or code. This document refers to two classes of segments: virtual and actual. (The term virtual, as applied to segments, is not to be confused with the ordinary use of the term "virtual memory"; an actual segment may or may not be present in main memory.)

Virtual segments are the objects manipulated by user programs; they are defined and referenced by special words called Data Segment Descriptors (DDs) and Code Segment Descriptors (CSDs). The term Virtual Segment Descriptor (VSD) refers to the union of DD and CSD. VSDs occur within the addressing environments of user programs, where they can be accessed and manipulated in limited ways.

Actual Segments and Actual Segment Descriptors (ASDs)

An actual segment is a contiguous group of memory words.

An actual segment is defined by a special object called an Actual Segment Descriptor (ASD). An ASD is a multi-word data structure that specifies the location, length, and other attributes of an actual segment. All the ASDs in the system are in an array called the ASD table; each entry has a unique index called its ASD_number. Once "touched", each VSD contains the ASD_number of the actual segment to which it refers.

The ASD table is outside the addressing environment of user programs; such programs refer to ASDs indirectly, by evaluating VSDs and other items that contain ASD references, but they do not manipulate the ASD_number references or ASD entries directly.

Virtual Segments and Virtual Segment Descriptors (DDs and CSDs)

Virtual data segments may be unpaged (represented by one actual segment of arbitrary length) or paged (subdivided into noncontiguous fixed-length actual segments, with possibly a shorter last page). Code segments are always unpaged. To an unpaged virtual segment there corresponds exactly one actual segment; in this case the virtual-actual distinction is not significant for the segment, but remains so for the descriptors, VSD and ASD. There are four classes of VSDs:

untouched	A request-token for a segment (or sometimes for a reference or other object). This is the only form of VSD constructed by user programs.
original	The defining and owning token for a segment. Touched original VSDs are created only by the operating system.
unindexed copy	A reference to the segment. Copies are created by operators that fetch original DDs from memory. Copy CSDs do not occur in user programs.
indexed copy	A reference to a specific element of the segment. Indexed DDs are always copies; they are created by Index operators from unindexed (original or copy) DDs. No indexed CSD is defined.

An untouched VSD could be called *untouched original*. It typically serves as a request for an original DD or CSD, but it is actually a general *binding-request token* and can occur wherever a DD or reference is expected. Any normal access to a token is intercepted so that the requested object can be created in place of the token.

The other three classes of VSDs are touched; they contain fields to locate and characterize the data in the segment. The element_size field of a DD indicates whether the virtual data segment is to be treated as an array of single words, double word pairs, or 4- or 8-bit (hex or EBCDIC) characters. All touched VSDs contain an ASD_number. The adjectives *original*, *copy*, *unindexed*, and *indexed*, when applied to a DD, imply touched; indexed also implies copy.

Generally, there is one original VSD for a segment. However, functional operator definition does not require precisely one original DD for each array.

Arrays of 4-bit characters are called hex because four bits can encode one hexadecimal (base 16) digit. Arrays of 8-bit characters are called EBCDIC for Extended Binary Coded Decimal Interchange Code, the character encodings commonly used on A Series systems. Certain operators associate particular hex or EBCDIC encodings with decimal digits or with + or - signs, but most character operators accommodate arbitrary 4- or 8-bit data.

Procedures: Activation Records

A *procedure* is a portion of a program, associating a group of parameters and declarations with a body of code. (This notion of procedure is consistent with the PROCEDURE or BLOCK of ALGOL, the PROCEDURE or FUNCTION of Pascal, and the SUBROUTINE or FUNCTION of FORTRAN.)

An *activation record* (AR) can be considered the instantiation of a procedure; it is a group of consecutive words that provide space for the data and control objects occurring in the procedure. Each word (or double word item) corresponds to one object declared in or passed as a parameter to the procedure. Simple objects such as INTEGERs, REALs, or BOOLEANs occur as operands within the record, while more complex structures, such as ARRAYs are defined by descriptors in the record. Other word types occur as references. The declarations within a procedure may include other procedures, which are generally represented in the AR by Program Control Words (PCWs) defining their entry points.

A *program* has a static hierarchical structure, in which each procedure occupies a fixed position. Objects declared within a procedure are considered *local* to it; they are instantiated as items in its AR. When procedure P declares procedure Q, Q is said to be nested within P. Conversely, P is said to contain Q, and items declared in P are considered global to Q. This characteristic of block structured languages is modeled in the A Series; the activation record of P is considered the immediately global activation record to that of Q. Procedure Q could, in turn, declare another procedure, R, and so on. Then Q is immediately global to R and P is more distantly global to R.

Each visible object can be named by specifying the nesting depth of the declaring procedure and the relative position of the item in that procedure's activation record. Such names are called *address couples* and are the sole form of address appearing in the code of A Series programs. An address couple is written (lambda,delta), where

lambda the *lexicographic level* (nesting depth, usually abbreviated *lex level*)

delta the position of the word in the activation record (displacement from the base of the AR).

Processes: Stacks

A *process* is an execution of a program; it associates the program code with the data space and processing record for that execution (and separate from all other executions). A new process involves the activation of a single procedure; in the course of its execution, that procedure can activate procedures, which can activate procedures, and so forth. Only the most recently activated procedure in the process executes code; when one procedure calls another, the caller is suspended until the called procedure exits. (Not all the procedures activated in a process are necessarily from the same program.)

Unlike procedure activation, which is performed automatically by the Enter operator, process creation is performed by operating system software. A process is terminated (also by software) when the initial procedure exits or is aborted.

A stack can be considered the instantiation of a process. The stack contains a historical record of all procedures that have been entered and not yet exited. It contains the activation records for those procedures. It also serves as an expression stack; most operators take their arguments from the top of the stack and leave their results there. For each procedure that has been entered in a process and not yet exited, there is one *frame* in the process stack. The stack frame includes the activation record followed by any items that have been pushed onto the expression stack and not yet consumed.

The system can use many stacks, with a processor assigned to one stack at a time (multiprogramming). A system can have more than one processor; at any time, each processor is bound to a different stack (multiprocessing).

A stack is a special case of actual segment. Stacks are referenced by *stack_numbers*; a stack_number is an index into a table of descriptors that define the stacks. On MCP/AS (Extended) systems the stacks are defined by ASDs and the stack_numbers are a subrange of the ASD_numbers. Stacks are identified as such in their defining ASDs.

Programs: Code Segment Dictionaries

Executable code is contained in segments defined by Code Segment Descriptors, which occur in special segments called *Code Segment Dictionaries*. A code segment dictionary is a special case of an activation record, global to the outer block AR. The code segment dictionary may be considered the instantiation of a program; its AR is global to an arbitrary number of processes executing the same program. In addition to a CSD for each code segment in the program, this AR contains the Program Control Word (PCW) defining the outer block entry point; it may also contain Data Descriptors defining read-only data segments. In order to execute a program, software creates a code segment dictionary associated with the program code file.

Linkages and Addressing Environments

Activation records are linked onto two lists: the historical chain and the environmental chain, which are at the base of each activation record. Both links are contained in the Mark Stack Control Word (MSCW).

The *historical chain* is a chronologically ordered list that consists of History Links connecting each stack frame to that of its initiating procedure (immediately below it in the same stack). A processor register (F) holds a pointer to the head of this chain for the active stack.

The *environmental chain* defines the current addressing environment of a process by linking the activation record of each procedure to that of its containing, or immediately global, procedure.

A normal Environment Link has two elements: stack_number and displacement, specifying a stack and the word index within the stack segment to the base of an activation record. A pseudo Environment Link is also defined; it has two elements: ASD_number and displacement, specifying an arbitrary segment and a small index.

An AR constructed automatically on a stack by procedure entry is linked into the environmental and historical chains. It contains a Return Control Word (RCW) that preserves state necessary to resume processing at the activation record designated by the History Link. Such an AR is called a normal activation record to distinguish it from a pseudo activation record, which is created in an arbitrary segment by primitive software. A pseudo AR is linked into the environmental chain, but never as the topmost entry. This AR is not linked into the historical chain and it has no RCW. A stack can contain an arbitrary number of normal ARs, but a pseudo AR typically encompasses all or most of an actual segment.

The current addressing environment is the set of activation records addressed by the environmental chain whose head is the current topmost AR, the one for the procedure whose code stream is currently bound to the processor. The position of an AR in the environmental chain defines its lex level. The lex level of the topmost AR is defined to be LL; there are LL+1 ARs in the current environment. Lex level 0 defines the end of the chain and denotes the most global AR. The processor maintains the current value of LL in a register.

A processor register (D[LL]) points to the base of the current topmost AR and, therefore, to the top of the environmental chain. There is another register pointing to the immediately global AR, and so forth to the end of the environmental chain. These registers are called display registers and are treated as a vector; the most global AR is referenced by D[0], the next by D[1], and so forth up to D[LL]. Display registers are used to evaluate address couples. The base of the activation record for level lambda is designated by D[lambda]. (In principle, address-couple evaluation can be performed by traversing the environmental chain when lambda is less than LL, but satisfactory performance requires the use of display registers.) The registers are maintained automatically by the processor, which traverses the new environmental chain at procedure entry/exit and process stack switch.

A History Link always points to an MSCW in the same stack; the historical chain is a linear structure that maps the sequence of procedure entries within a process. An Environmental Link may point at an MSCW in the same or another stack; the set of environmental chains forms a tree structure that maps the static block structure of a program. The historical and environmental chains are often called *dynamic* and *static*, respectively. Early versions of the system architecture used the term lexical for the environmental chain; the term lexicographic occurs in the literature of block structured programming languages, referring to the static structure of the program; hence, the phrase lexicographic level.

References

References to objects occur in several forms:

1. An *address couple* in the code stream is consumed directly by an operator (such as Value Call, which fetches an operand). Address-couple evaluation depends upon the current addressing environment (context), because its lambda component specifies only a position on the current environmental chain.
2. A *Stuffed Indirect Reference Word (SIRW)* is a context-independent reference formed on the A16 by transforming an address couple with a Name Call operator, which replaces the lambda component of the address couple by an environment link to the activation record at lex level lambda in the current addressing environment. An SIRW is created from an address couple to designate an item in the current environment, but once formed it continues to designate that item when passed into other environments.
3. An *unindexed copy DD* is a reference to a segment. Such copies are formed from original DDs by Load operators.
4. An *indexed copy DD* is a reference to an element of a segment; it may also be used to designate the first element of a sequence. Indexed DDs are formed from unindexed DD by Index operators.
5. A *Program Control Word (PCW)* constitutes a specialized reference to a code stream. PCWs define procedure entry points; they also designate destinations for jumps between segments. A PCW is generated by a special Make PCW operator from a literal in the code stream. These MPCW literals contain the only code stream references to the address couples of Code Segment Descriptors.

   A PCW also contains other information about the code, including the lex level at which an entered procedure will run.
6. An *Actual Store Reference Word (ASRW)* is a special reference used by a few operators to access memory by absolute address. ASRWs are not manipulated in user programs; they are mentioned here because they appear in the definitions of some partly restricted operators. That is, the operators occur in user programs and the operators can consume an ASRW, but ASRWs do not occur in user programs.

Protection

The term protection refers to mechanisms and techniques that control the access of executing programs to stored information; protection is one aspect of the much larger topic of security.

The protection mechanisms in A Series processors can be described as a *modified capability architecture.* A capability is defined as an unforgeable ticket that grants the possessing subject (process or procedure) the right to access a designated object (data) in specified ways (such as read, write). (The term "modified" denotes that there are restrictions on the access to and manipulation of the capabilities themselves.)

The remainder of this section is a brief overview of the protection mechanisms for MCP/AS (Extended) systems.

The fundamental protection policy is that a procedure can access only the objects in its extended addressing environment. The access modes of primary interest are read, write, and enter; additional accesses include create, destroy, and exit. A procedure creates objects by pushing words onto the stack, including operands that define simple objects and untouched DDs that define more complex ones. Exiting a procedure destroys all the objects declared in it; the RCW serves as an exit capability (usable once only). The environmental chain provides, via address couples, read and write capability to objects in the addressing environment. An SIRW or DD provides read and usually write capability to the referent. A PCW serves as an enter capability.

Protection is a responsibility shared by the processor (hardware and firmware) and the software. The responsible software includes the user language compilers, which are constructed to follow a number of conventions in all generated code. In particular, the unforgeability property of capabilities is enforced by the compilers, which do not emit sequences of code that could create or modify a capability contrary to the policy. The compilers are also "trusted" to generate only valid address couples.

Accessing Segments

Enforcement of segment boundaries is a key element of the protection mechanism. Data in segments (other than from ARs) is accessed via IndexedDDs; enforcement is based on controlling the generation and use of IndexedDD:

1. IndexedDDs are formed by index operators, which apply an index value to a DD. Both unindexed and indexed DDs can be indexed; reindexing an IndexedDD algebraically adds the new index to the current index. The index operations require resulting index values between zero and the index-field limit of the IndexedDD. A DD can be indexed by the array length, generating an IndexedDD that indicates the first element beyond the segment. All index bound enforcement is at word boundaries.

Indexing a paged DD proceeds in two steps:

a. The index is divided by the page length .

b. The quotient is used to index the page table and fetch the unindexed page DD, which is then indexed by the remainder from the division.

The upper bound on unpaged arrays is determined by the length of the segment; the upper bound on paged arrays is determined by the length of the page table, the fixed page length, and the length of the last page. Indexing a paged unindexed DD proceeds in two steps, as described previously; the quotient must be a valid index on the page table. Reindexing a paged indexed DD is more complicated, but the result is as though the sum of the current and new index values were applied to the unindexed DD.

2. Load and store operators consume an IndexedDD to access the reference word or double-word. Load and store operations require the word index in an IndexedDD to be less than the segment length specified in the referenced ASD.

3. Array operators use an IndexedDD to access one or a sequence of elements of an array; they may modify the IndexedDD while traversing the array. Array operators are bounded by the length in the ASD.

4. Certain operators make repeated read access to a membership or translate table, indexing an IndexedDD by a function of the character values in a sequence. Table access is bounded by the length in the ASD.

The boundaries of code segments and edit tables are enforced statically by the compilers. The A16 enforces those boundaries for all executed code, based on the length in the ASD.

Access Control

In their roles as capabilities, the descriptors, references, and other control structures offer access rights to the holder. Some of these rights can be controlled individually for each instance of the structure.

An indexed or unindexed DD has a write-access right that can be disabled. Write access is required for any memory write (store) operation. A write-disabled DD is often called read only. The default state is write enabled.

An IndexedDD has a reindex right that can be disabled. Reindex must be enabled for the IndexedDD to be used as an argument to an indexing, array, or set-element_size operator. The default state is reindex enabled.

Addressing

The only addresses that appear in a code stream are static address couples (lambda,delta). The maximum valid values of lambda and delta are respectively LL and the offset of the last item declared in the referenced activation record.

Simple variables, references, and other one and two word objects reside in an activation record and can be addressed directly by an address couple. When the addressed object is a reference (SIRW or copy DD), it can be evaluated to provide access to its referent.

Larger memory objects, such as arrays and multi-word structures, typically reside in segments outside the stack and are defined by original DDs within the stack or within other segments. An element of an array declared in a procedure can be addressed by providing an integer index value and the address couple of the descriptor. If that element is an indexable descriptor, it can be indexed, and so forth.

Code is addressed by PCWs. In general, the entry point of a procedure is specified by a reference to its PCW and the activation record in which the PCW occurs becomes the immediate global addressing space for the new AR instantiating the procedure. Also, the PCW itself (rather than the reference) can be used to enter a procedure whose immediate global AR is within the current addressing environment.

The term *addressing environment* refers to the set of activation records that constitute the current procedure and its globals. The items in that environment are directly visible via address couples. Other items, arrays, elements, and code are indirectly visible because references or original DDs for them are visible. These indirectly visible objects may themselves contain referenced or original DDs that render other objects indirectly visible, and so on. The union of the directly and indirectly visible objects constitute the extended addressing environment of the procedure.

Procedure Entry/Exit

Normal activation records are created on the top of the stack by a sequence of operations that effect procedure entry:

1. A Mark Stack operation defines the base location for the beginning activation record, creating a new MSCW on top of the stack and linking it at the head of the historical chain. At this step the MSCW is marked "inactive"; a history, but not an environment, link exists and the incipient AR is actually within the expression stack of the initiating process.
2. A reference to the code for the procedure is placed on the stack.

3. Any parameters for the procedure are placed on the stack.
4. An Enter operation is performed. The new AR is now linked into the environmental chain; the addressing environment and code stream for the new procedure are established. The reference to the new procedure is replaced by a special structure called a Return Control Word (RCW), which designates the invoking code and preserves other states necessary to resume the invoking procedure when the invoked procedure is completed (exits).

   At this point the MSCW is marked "entered" (the opposite of "inactive") and the new AR formally exists.
5. The new AR may now be extended as the prologue (stack-building) code in the new procedure pushes objects onto the stack. Such objects may include operands to define simple scalar objects, untouched descriptors to define additional segments or binding requests, PCWs to define nested procedures, and so forth. At first, these objects are in the expression stack; an explicit Push operator must be executed to make the new objects accessible via address couples. An Enter operation implicitly causes a push to display the parameter words already on the stack, which occupy the low order part of the new AR.

In explicit procedure invocation, a program performs each of the previous steps by executing appropriate operators. In addition to explicit procedure entry, there are two mechanisms in which entry is implicit; in these cases, the relevant Steps 1 through 4 occur automatically:

- An interrupt is an entry (into an operating system procedure) that is performed automatically by the processor because an operator is unable to perform its function or because some condition requires software action.
- An accidental entry occurs when a PCW is encountered during the evaluation of a reference chain. The designated procedure is invoked as a function with no parameters; it is expected to return either another reference or the target of the reference chain. This mechanism is provided to support the call-by-name parameter semantics of ALGOL-60. The invocation of an accidental entry procedure is often called a *thunk*.

The topmost stack frame is deleted from the stack by an Exit operation; the historical chain is used to locate the prior entered AR and reinstate its stack frame and addressing environment, and the RCW is used to resume the code stream.

Expression Stack and Stack Frames

The current expression stack occurs at the current top of the process stack, immediately above the topmost activation record. A processor register (S) maintains a pointer to the top of the current stack. (The topmost several words of the stack may themselves be in registers; the remainder of the stack is in main memory.)

If a procedure is invoked when the current expression stack is not empty, the expression values remain in the stack below the new AR. Upon procedure exit, the topmost expression stack is discarded along with the topmost AR, and the previous stack is reinstated.

The term *stack frame* is applied to the combination of the activation record and the expression stack of a procedure. The distinction is that address couples and SIRWs can designate only items in the activation record. The boundary between the expression stack and the topmost activation record is determined by the length of the activation record. A Push operation enlarges the activation record to include the expression stack.

The distinction between activation record and expression stack is maintained by convention. The processor enforces no distinction, but the results of addressing a word in the expression stack are undefined.

Stack and Display Registers

The processor maintains several registers that designate the current stack, historical chain, and addressing environment.

- **SNR** contains the stack_number of the stack to which the processor is currently bound. This value serves as a unique process identifier throughout the life of the process; it is available to user programs for that purpose.
- **S** designates the top of the expression stack.
- **F** designates the topmost MSCW, the head of the historical chain.
- **D[lambda]** designates the base of the activation record at lex level lambda in the current addressing environment.
- **D[LL]** designates the topmost entered MSCW (at the base of the topmost activation record), the head of the environmental chain.

Some, but not all, of these registers are accessible directly by programs. When accessed by programs, S, F, and D[LL] are defined for MCP/AS (Extended) systems to contain displacements relative to the base of the stack designated by SNR. The obsolete register BOSR is defined as a constant zero, so that differences such as S-BOSR are invariant across all A Series systems.

Executable Code Streams

Variable length operator sequences are stored in arrays of program code words called *code segments*. Each program code word contains six 8-bit containers called *syllables*.

The term *code stream pointer* is used to describe a reference to the entry point of an operator sequence in a code segment. A code stream pointer consists of a reference to the code segment and indexes to a word and syllable within the segment.

For MCP/AS (Extended) systems, the code segment reference is the ASD_number of the segment. The processor retains this number in a register, Program Segment Number (PSN).

The entry point in the code segment is indicated by the Program Word Index (PWI), which is relative to the base of the code segment, and the Program Syllable Index (PSI) within that word. The processor maintains PWI and PSI in registers.

Processors retain additional information about the code stream:

- The **LL register** holds the lex level of the current procedure.
- The **CS Boolean** indicates whether the processor is in normal state or control state. In control state, the processor defers most asynchronous interruptions of the code stream until a normal state is resumed. User programs almost always run in a normal state.

General Boolean Accumulators

Processor state includes several Boolean accumulators that are used by several operator groups. Their use and definition are discussed in Section 3 and are summarized in Appendix B.

TFFF:	True/false flip-flop.
OFFF:	Overflow flip-flop.
EXTF:	External sign flip-flop.
FLTF:	Float flip-flop.

A variety of operators interact with TFFF, OFFF, and EXTF; the values of these registers are automatically saved and restored by procedure entry and exit. FLTF is used only within sequences of special edit operators. Its value is preserved as necessary when these sequences are interrupted and resumed.

ASD Memory Architecture

Each memory word is directly named by an absolute address. For the A16, an absolute address is an integer in the range 0 to 2**29-1.

System memory is subdivided into contiguous regions, called *segments*, which are defined by ASDs. Direct access to absolute addresses is limited to low level system software; ordinary references to memory are accomplished indirectly, by citing a segment (ASD_number) and a segment-relative offset; the address is derived from the ASD for that segment. (Even references via address couples are ultimately defined by ASDs.)

All the ASDs in the system are located in a structure called the ASD table. For MCP/AS (Extended) systems, the table is at a fixed location, which starts at absolute address = 0.

Section 2
Data Structures and Types

Most data types are defined in this section; others are defined later in the context of the operators involved. Except where noted, the definitions in this section apply to all MCP/AS (Extended) systems.

Words have 48 information bits plus four tag bits. The information bits are numbered from 47 to zero, from the high-order bit down to the low-order bit. Word formats are conventionally diagrammed with the bits shown in groups of four and the tag separated from the 48 bit body of the word:

t a g	47	43	39	35	31	27	23	19	15	11	7	3
	46	42	38	34	30	26	22	18	14	10	6	2
	45	41	37	33	29	25	21	17	13	9	5	1
	44	40	36	32	28	24	20	16	12	8	4	0

Figure 2-1. Word Format

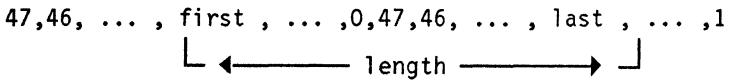
A field of a word is denoted [first:length], where

first is the bit number of the high-order bit of the field (first less than or equal to 47)

length is the field length in bits (length less than or equal to 48).

Fields are often given names, such as "offset" for "[12:13]". A field of an object or the class formed by applying the field specification to a class of objects is denoted by suffixing the field name or specification to the object or type name with an interposed dot, such as x.[46:1] or SIRW.offset.

When necessary, fields are wrapped around from the lowest order bit to the highest order bit. If length is greater than first+1, then length minus (first+1) bits are concatenated starting from the highest order bit (47). The following sequence illustrates the field [first:length] in the case where length is greater than first+1:



Word types are distinguished by tag value and frequently by additional type bits in the word. Tag values are specified using hexadecimal notation.

Throughout this document, word types are referred to by type name or acronym. Because the data type double precision consists of two words, the term *item* is used (instead of *word*) to refer to an entity whose type may be double precision.

Data Words

A *Data Word* is defined as a word that, when present in memory, can have its contents rewritten by a normal store (as opposed to overwrite) operation. (Words of all other types are protected from normal stores.) Data Words serve primarily as computation arguments, rather than as reference arguments or control structures.

Data Words include those with tags in {0, 2, 4, 6}.

The term Data Word denotes a union of data types; these types are defined in the following paragraphs. Their definitions are the same for all A Series systems.

Operands (Single and Double Precision)

The great majority of data items dealt with by programs are operands. There are two operand types:

single precision:	a single word with a tag of 0
double precision:	a pair of words both with a tag of 2

Throughout this document, the unqualified term *operand* is used solely to refer to the type union single and double precision.

Neither operand type has a unique interpretation applied to it. Operators, according to their function, apply to different interpretations. For example, operands are interpreted as numeric values, bit vectors, and character sequences by arithmetic, word manipulation, and pointer operators, respectively. Numeric and Boolean operands are generated and interpreted by a wide variety of operators and are defined in this section. Section 3 defines additional operand interpretations with the operator groups that apply them.

The terms *single* and *double precision* convey the usual notion of alternate numerical representations with different precision and range, as defined in the following paragraphs. However, the same terms define data types regardless of the interpretation being applied.

Real Numeric Operands

Many operators interpret operands as numeric values. The structure of the numeric data is defined in this section; details of numeric interpretation are found in the operator descriptions under "Numeric Operand Interpretation" in Section 4.

A single precision floating point operand is represented as a single precision operand with the following fields:

	[47: 1]	Not used
mant_sign	[46: 1]	Mantissa sign (0=positive, 1=negative)
exp_sign	[45: 1]	Exponent sign (0=positive, 1=negative)
exponent	[44: 6]	The power of eight by which the mantissa is scaled
mantissa	[38:39]	The integer magnitude of the number before scaling

The mantissa field can be regarded equivalently as 39 binary digits or 13 octal digits; the latter view is convenient because floating point numbers are scaled by powers of eight.

A single precision floating point operand with an exponent equal to zero is used as the canonical representation of a single precision integer; an operand in this form is called *single_integer*. The largest single_integer magnitude is 2**39-1 = 8*13-1.

The form "k-bit integer" (where k is an integer in {1 to 39}) is used to specify an operator output value in which field [47:48-k] contains zero. The same term is used to specify an operator input value in which fields [46:1] and [44:45-k] contain zero or field [44:45] contains zero. (Bits 47 and 45 are insignificant in integer representations; bit 46 is insignificant if the value is zero.)

A double precision floating point operand is represented as a double precision operand with the following fields:

```
             [47: 1]     Not used
mant_sign    [46: 1]     Mantissa sign (0=positive, 1=negative)
exp_sign     [45: 1]     Exponent sign (0=positive, 1=negative)
exponent     [44: 6]     The low order 6 bits of the exponent
mantissa     [38:39]     The integral portion of the mantissa

Second word:

hi_order_exp [47: 9]     The high order 9 bits of the exponent
mantissa     [38:39]     The fractional portion of the mantissa
```

A double precision floating point operand with an exponent equal to thirteen is used as a canonical representation of a double precision integer; an operand in this form is called a *double_integer*. (A double precision operand with exponent and mantissa zero is also a valid double_integer.) The largest double_integer magnitude is 2**78-1 = 8*26-1.

Decimal Sequence Operands

Some operators generate or consume operands interpreted as a sequence of decimal digits. Each digit is represented as a four bit binary integer in the range {0 to 9}. (Hex sequence representations of decimal data are often called Binary Coded Decimal, BCD.) A *digit sequence* is an unsigned sequence of decimal digits. Digit sequences generated by binary-to-decimal conversion and scale right operators contain only true digits; however, digit sequences that are inputs to pointer operators may include nondigits, which are hex characters in the range {hex "A" to hex "F"}. Up to 12 or 24 digits can be contained in a single or double precision operand, respectively; the second word of a double holds the low order digits. Depending on the operator, the digit sequence is left- or right-justified within the operand.

Boolean Operands

Some operators generate and other operators consume operands interpreted as Boolean values. The Boolean values true and false are represented as binary 1 and 0, respectively. The terms True and False are used to specify Boolean values. In the specification of a stack output from an operator, True and False are defined as the 1-bit integer values 1 and 0, respectively. In the specification of a stack argument for an operator, True and False are defined as operands with bit [0:1] equal to 1 or 0, respectively; Boolean interpretation ignores field [47:47] of any operand and the second word of a double precision operand.

Boolean operands are generated by the relational operators, among others. Operands are interpreted as Boolean values by the branch operators and Assert operators. The logical operators do not interpret Boolean values; rather, they perform Boolean arithmetic on the argument bit vectors.

Tag-4 Word and Tag-6 Word

Tag-4 and tag-6 words are data words, but the only interpretation applied to them is as 48-bit vectors by a class of computational operators. A zero word with a tag of 6 is conventionally used as an *uninitialized single datum*.

These words cannot normally be fetched to the expression stack as operands and they are not valid arguments for arithmetic computational operators. However, they may be fetched with the LOAD operator and may be stored over by normal store operators.

Program Code Words

Variable length operator sequences are stored in arrays of program code words called *code segments*. Each program code word contains six 8-bit containers called *syllables*, numbered 0 to 5 from high order to low order. (The mapping of each operator into syllables is specified in Section 4.)

```
s[i] = [47-(8*i): 8]  Syllable i, for i = 0 to 5
```

Program code words have tags of 3, except for special sequences called Edit Tables, which have tags of 0.

Actual Segment Descriptors (ASD)

An ASD is a four-word structure; the constituent words are called ASD1, ASD2, ASD3, and ASD4.

The ASD table is a two dimensional array containing four words per ASD: ASDi[j] is defined for j (ASD_number) in {0 to 2**23-1} and i in {1 to 4}. The table is stored in memory as a linearized array beginning at absolute address zero:

```
ASDi[j] = Memory[4*j + i-1]
```

ASD[0] is the first 4-word entry in the table. It may be accessed with the RASD and WASD operators, but 0 is not a valid ASD_number for a reference. Processors are not required to detect the use of 0 as an ASD reference, so software should never create a valid ASD[0].

ASD1 contains the absolute memory address of the first word of the actual segment (if the segment is present in memory) and several other bits of information concerning the status of the segment.

ASD2 defines the length of the actual segment.

ASD3 may contain a reference to the Virtual Segment Descriptor that defines the virtual segment corresponding to this actual segment. For a paged virtual segment, the ASD3 for each page must be a reference to the page descriptor in the page table.

ASD4 has no defined function in the processor; it is reserved for software use.

Actual Segment Descriptor, First Word (ASD1)

Memory management of the actual segment uses two presence indicators and an address field. ASD1.present_to_proc must be 1 for any access to the contents of the segment by a data processor or any access by an I/O processor, except I/O data transfer. ASD1.present_for_IO must be 1 for any use of the segment as the source or destination for I/O data transfer. If ASD1.present_to_proc or ASD1.present_for_IO is 1, ASD.address is the absolute address of the first word of the segment; otherwise, the address field contains a software-encoded value.

ASD1 (Tag = 8)

	[47: 2]	Reserved for coexistence
present_to_proc	[45: 1]	Segment-is-present indicator for non-I/O access (0=absent, 1=present)
	[44: 1]	Reserved for coexistence
hlt	[43: 1]	Enables switch-dependent halt when executing code from this segment
not_stack	[42: 1]	Stack indicator (0=stack, 1=not stack)
present_for_IO	[41: 1]	Segment-is-present indicator for use as an I/O buffer (0=absent, 1=present)
	[35: 4]	Reserved for software use
address	[31:32]	If present_to_proc and present_for_IO are both zero, the address field is interpreted by software; otherwise, it contains the base address of the actual segment

Actual Segment Descriptor, Second Word (ASD2)

An ASD2 specifies the length of the segment; the remainder of the word is reserved for software use. AS_length specifies the actual-segment length for all operations that access data in the segment. The deficit and nybbles fields are used (in addition to AS_length) by the GLEN operator to report the user-declared length of the segment.

ASD2 (Tag = 8)

	[47: 8]	Reserved for coexistence
nybbles	[39: 4]	4-bit frames in addition to net words
deficit	[35:16]	Words less than AS_length-derived length
AS_length	[19:20]	Actual-segment length in words.

Absolute Store Reference Word (ASRW)

An ASRW references an individual item in memory via an absolute address. ASRWs are recognized by a restricted set of operators: LODT, OVRD, OVRN, RDLK, and TWOD.

```
ASRW (Tag = B)

           [47: 8]      Reserved for coexistence
address    [31:32]      Absolute address
```

Virtual Segment Descriptor (VSD)

Virtual Segment Descriptor (VSD) refers to the union of word types that describe virtual segments. A VSD is called a DD or CSD when used to define a data segment or a program code segment, respectively, but there is not always a distinction in the form or structure of the descriptors.

The following paragraphs describe the attributes of VSDs that are necessary for functional definition of the operators; not all of these attributes are subject to direct manipulation in user programs. Code in user programs manipulate only those fields that have uniform structure and interpretation across all A Series systems.

Untouched DD

An untouched DD is encoded as a word with a tag equal to five and field [47:2] equal to zero. (All tag 5 words are untouched DDs; [47:2] is available for system use, but not for encodings that must be common with all other A Series systems.)

By convention, the fields within [45:46] of an untouched DD correspond to those in a touched original, except that the ASD-reference field contains a software encoded value and the element_size is seven when the requested object is not an ordinary data segment (array). However, the contents of an untouched VSD are determined more by software convention than by operator functional definition.

Unindexed (Original and Copy) DD

```
Unindexed (original or copy) DD (Tag = C or E for unpaged or paged, respectively)

copy          [44: 1]   Copy bit (0=original, 1=copy)
read_only     [43: 1]   Read-only bit (0=read/write, 1=read-only)
element_size  [42: 3]   DD: array element type (0=single precision, 1=double
                         precision, 2=hex, 4=EBCDIC; 3,5,6,7 are invalid)
                        CSD: Reserved for software
              [39:20]   Reserved for software
ASD_ref       [19:23]   ASD_number
```

Note: *For an unindexed paged DD, the ASD_number is that of the page table. (Unindexed DDs for the pages occur in the page table, but the user program has no direct access to them.)*

A touched CSD is identical to a read-only unpaged original DD, but the element_size is not significant to the processor. By convention, touched CSDs are originals, however, this rule is not enforced by the processor.

The terms *word descriptor* and *WordDD* are used for descriptors whose element_size values are single or double (precision); the terms *character descriptor* and *CharDD* are used for descriptors whose element_size values are EBCDIC or hex. The terms *SingleDD* and *DoubleDD* are used for WordDDs with element_size single and double, respectively.

The no_write (read_only) bit in a DD can be set to prevent use of the DD for write access to the data.

The following fields are defined for use in user code files. User programs may set, but should never reset, the read_only bit.

```
read_only       [43: 1]
element_size    [42: 3]
```

The length of a segment is determined on older systems by examining the length field in an unindexed DD. The MCP/AS (Extended) systems require that the GLEN <get length> operator must be used instead. Programs that are to run on earlier systems must use selected code sequences and/or operating system interfaces to query segment lengths.

IndexedDD (Indexed Data Descriptor)

Indexed copy DDs are called IndexedDDs.

```
IndexedDD (Tag = D or F for unpaged or paged)
reindex       [44: 1]     Controls indexing, Data Array, and set-
                          element_size operators (0=disabled, 1=enabled)
read_only     [43: 1]     Read-only bit (0=read/write, 1=read-only)
element_size  [42: 3]     The type of array element (0=single precision,
                          1=double precision, 2=hex, 4=EBCDIC, and {3, 5,
                          6, 7} are invalid values)
index         [39:20]     Used if element_size is single or double
                          precision: the word index from the base of the
                          array to the referenced item
char_index    [39: 4]     Used if element_size is EBCDIC or hex: the index
                          within the word of the referenced character
word_index    [35:16]     Used if element_size is EBCDIC or hex: the index
                          from the base of the array to the word containing
                          the referenced character
ASD_ref       [19:23]     ASD_number
```

The following fields are common to indexed and unindexed DDs and function identically: read_only, element_size, and ASD_ref.

The reindex bit indicates whether the IndexedDD grants access to just the array element it references or to the entire segment. If reindex is disabled in an IndexedDD, the indexing operators cannot reindex it, the array operators cannot use it, and operators cannot change the element_size.

An indexed word descriptor is called an IndexedWordDD or, more specifically, an IndexedSingleDD or IndexedDoubleDD. An indexed character descriptor is called an IndexedCharDD. These terms are sometimes abbreviated IWDD, I1DD, I2DD, and ICDD.

The interpretation of the index to the referenced element depends on element_size. For IndexedWordDDs, the index field is the word index from the base of the array or page to the single precision word or to the first word of the double precision word pair. For IndexedCharDDs, the index field consists of two subfields: word_index, the index from the base of the array or page to the word containing the referenced character, and char_index, the character index within the word. Char_index must not exceed 5 and 11 for EBCDIC and hex IndexedCharDDs, respectively. Char_index 0 designates the highest-order character in the word.

The following fields are defined for use in user code files. User programs may set, but should never reset, the read_only bit.

```
read_only      [43: 1]
element_size   [42: 3]
char_index     [39: 4]     Used if element_size is EBCDIC or hex
```

Code Segment Descriptor (CSD)

CSDs occur in untouched and touched original forms, which are analogous to untouched and original DDs.

An untouched CSD is a word that occurs in a code segment dictionary with a tag of 3 and field [47:2] equal to zero. This encoding is not necessarily unique, but other words so encoded must not appear in a code segment dictionary where they may be the target of the (sdi,sdll) address couple in a PCW skeleton.

Touched CSDs are unpaged original DDs (defined previously) with write access disabled; the element_size field is reserved for software. Code and data segments are not interchangeable, however, because data-fetching operators cannot access code words and processors cannot execute data words.

Virtual Segments

An unpaged virtual segment is a single actual segment; it cannot exceed 2**20-1 words in length. An IndexedCharDD can designate only the first 2**16 words of an unpaged segment.

A paged virtual segment is represented by several actual segments: a page table and one or more pages. The page table contains an untouched or original DD for each page. Any unindexed DD for the segment has a special tag designated "paged" and refers to the page table. Any indexed DD for the segment designates an element of a particular page. The page DDs (the original DDs in the page table) are not marked paged, but the IndexedDDs designating elements in pages have a special tag.

The length in words of a full page is 2**13 for the A16 processor. All pages are 2**13 words long, except the last page in a virtual segment, which may be shorter.

The ASD for each page contains a reference to the original DD for that page; this reference is an IndexedSingleDD whose ASD_ref designates the page table and whose index is the page index. This back link from page to page table is essential for automatic page switching operations.

In summary, a paged virtual segment is a two layer structure; the DD for the whole segment specifies the page table, which contains a DD for each page. An unused page has an untouched DD; for each touched page, the DD refers to the defining ASD.

MaxLen, the maximum length of a paged virtual segment, is the product of the length in words of a full page times the maximum page table length. For the A16, this product is 2**13 * 2**15, which is 2**28.

Another parameter is called MaxInx; it is the largest integer value that can be accommodated in the indexing hardware and the iteration counts for pointer operators. MaxInx is an arbitrary limit that typically corresponds to a register width; it is at least as large as any useful word or character index value. For the A16, MaxInx is 2**32-1.

Virtual Index

Any word or character in a virtual segment can be specified by an integer called its *virtual index*. The virtual index of the element is the zero-relative ordinal of that element when the entire segment is considered as an array of elements of that size. An equivalent operational definition of virtual index is as an integer that can generate an IndexedDD designating the element when the integer is applied by the Index operator to an unindexed DD that designates the virtual segment and has the appropriate element_size.

This definition of virtual index can be extended to double precision elements only for elements with even word alignment within the segment. Because double precision elements can occur with odd alignment, it is more useful to define the virtual index of a double item as the virtual index of its first word.

The virtual index of an IndexedDD is stored in as many as three components:

1. The **page index (pX)**, for a paged segment, designates the page; it is an index into the page table. The pX value is implicit, because the IndexedDD is a copy of page[pX] of the page table. The ASD referenced by the IndexedDD contains a reference to the page descriptor; the reference is an IndexedSingleDD with pX in its index field. For an unpaged segment, pX is defined to be zero.
2. The **word index (wX)** designates the referenced word within the actual segment; wX is in the index field of an IndexedWordDD or in the word_index field of an IndexedCharDD.
3. The **character index (cX)** designates the referenced character within the word; cX is in the char_index field of an IndexedCharDD.

The virtual index (vX) for an IndexedDD is computed from these three values, according to the element_size of the IndexedDD:

single or double:	vX = pX*2**13 + wX
4 bit:	vX = (pX*2**13 + wX)*12 + cX
8 bit:	vX = (pX*2**13 + wX)*6 + cX

References

Of the various reference types, unindexed and indexed DDs are virtual segment descriptors and are defined previously. This section defines address couples, SIRWs, PCW literals, and PCWs.

Address Couples

Fixed Fence Address Couples

Address couples in several operators are encoded in 16 bits with a 4-bit lambda value in field [15:4] and a 12-bit delta value in field [11:12].

Variable Fence Address Couples

Address couples in NAMC and VALC operators are encoded in 14 bits with a *variable fence* between lambda and delta. Taking advantage of the fact that lambda must be less than or equal to LL, the number of high order bits interpreted as the lambda value varies with the value of LL at evaluation time. The remaining low order bits are interpreted as the delta value. The following table gives explicit ranges:

LL range	bits left of fence	lambda range	delta range
{0 to 3}	2	{0 to LL}	{0 to 2**12-1}
{4 to 7}	3	{0 to LL}	{0 to 2**11-1}
{8 to 15}	4	{0 to LL}	{0 to 2**10-1}

The lambda value is the reverse of the bits to the left of the fence; the delta value is taken from the bits to the right of the fence. Following are examples of address couple interpretation. Each pair is the same address couple representation, but notice the effect of the dynamic fence indicated by the colon (:).

1. a. at LL= 2, 10:000000010011 => (1,19)

   b. at LL=13, 1000:0000010011 => (1,19)

2. a. at LL= 5, 101:00001000000 => (5,64)

   b. at LL= 3, 10:100001000000 => (1,2112)

CSD Address Couples

Address couples of Code Segment Descriptors occur in PCW skeletons. These use a fixed fence form with lambda in [13:1] (called sdll) and delta in [12:13] (sdi).

Environment Links

An *Environment Link* specifies an activation record by identifying the segment that contains it and the number of words from the base of the segment to the MSCW at the AR base. User programs do not manipulate Environment Links, but they are fundamental to an understanding of addressing.

An Environment Link is a meta-field that is defined here as an object with several components, so that the entire Link can then be considered a single field in the words in which it occurs (SIRWs and entered MSCWs). An Environment Link has two possible formats:

normal Environment Link

stack_num	[46:15]	The stack_number identifying the stack containing the referenced location
stack_disp	[31:16]	The displacement from the base of the stack to the MSCW of the normal activation record
pseudolink	[15: 1]	0: denotes normal env_link

pseudo Environment Link

seg_num	[46:23]	The ASD_number identifying the segment containing the referenced location
seg_disp	[23: 8]	The displacement from the base of the segment to the MSCW of the pseudo activation record
pseudolink	[15: 1]	1: denotes pseudo env_link

Normal activation records are constructed automatically by procedure entry. Pseudo activation records are constructed only by privileged software, which also must forge pseudo links in SIRW references to PCWs placed within the record. Procedure entry via those references to those PCWs causes the pseudo AR to be linked into the environmental chain as the immediate global AR of the entered procedure; within the procedure (and any procedures local to it), SIRWs constructed to address couples within the pseudo AR will automatically have pseudo links.

Stuffed Indirect Reference Word (SIRW)

An SIRW is created (on the A16) by the NAMC and LNMC operators from an address couple, to provide a context-independent reference to the same location.

```
SIRW (Tag = 1)

env_link     [46:32]   Designates activation record in which reference occurs
SIRW_soft    [13: 1]   Reserved for software use
offset       [12:13]   The offset from the base of the AR to the referenced
                        location
```

SIRW.offset corresponds to address couple delta, but is one bit wider. For some purposes an SIRW must be able to span a code segment dictionary.

Together the env_link and offset fields address an item in an activation record. The item is located in the segment designated in env_link, indexed by the sum of the displacement in env_link plus the offset. Some operations, notably procedure entry, require both the displacement and offset, from which ordinary referencing uses the sum.

The following fields are defined for use in user code files:

```
SIRW_soft    [13: 1]
offset       [12:13]
```

Program Control Word (PCW)

```
PCW (Tag = 7)

                [44: 1]   Must be zero
psi             [43: 3]   The PSI code stream pointer component
pwi             [40:13]   The PWI code stream pointer component
control_state   [27: 1]   CS Boolean (0=normal state, 1=control state)
                [26: 1]   Must be zero
lex_level       [25: 4]   The lex level at which the activation record runs
                [21: 2]   Must be zero
ASD_ref         [19:23]   ASD_number
```

The PCW code stream pointer consists of the field pwi, psi, and ASD_ref. The lex_level field specifies the lex level of the activation record associated with the designated code. The control_state attribute specifies execution in normal and control state.

User programs do not manipulate PCWs. However, they do provide the skeletons from which PCWs are constructed by the MPCW operator. The following is the structure of the PCW skeleton.

```
                [47:12]  Zero
skeleton_psi    [35: 3]  The PSI code stream pointer component
skeleton_pwi    [32:13]  The PWI code stream pointer component
skeleton_CS     [19: 1]  The initial value of the CS Boolean
                         (0=normal state, 1=control state)
                [18: 1]  Zero
skeleton_LL     [17: 4]  The lex level for the procedure
sdll            [13: 1]  The lambda component of an address couple
                         designating CSD
sdi             [12:13]  The delta component of an address couple
                         designating CSD
```

Stack Structure and Linkage

The contents of a process stack include a top-of-stack control word at the base of the stack, followed by an arbitrary number of words used by software and then an arbitrary number of stack frames.

There are two data types used for stack frame linkage: a Mark Stack Control Word (MSCW) and a Return Control Word (RCW) appear at the base of each stack frame for linkage.

A Top of Stack Control Word (TSCW) is used to preserve processor state in an inactive stack (a stack to which no processor is bound).

These structures are not unambiguously typed; their recognition depends in part upon the context in which they are accessed.

With certain exceptions involving the RCW, user programs do not manipulate stack linkage words. Their attributes are defined in the following subsections for purposes of defining the operator function.

Stack Frame Linkage

There are two words (MSCW and RCW) at the base of an entered normal activation record. The MSCW is at offset 0 and the RCW at offset 1. No RCW is used with a pseudo AR.

The inactive MSCW is built by the Mark Stack operation:

```
Inactive MSCW  (Tag=1)

               [47: 1]  Zero
               [46:23]  Must be zero
history_link_I [15:16]  Self-relative displacement to prior MSCW
```

The entered MSCW and RCW are built by the Enter operation; they occur only in entered ARs:

Entered MSCW (Tag = 3)

exit_opt	[47: 1]	0: no optimization information 1: exit only updates D[new LL], except if new LL is greater than old LL, also updates down to D[old LL] (A16 definition)
env_link	[46:32]	Identifies the immediate global activation record (if lex_level greater than 0)
block_exit	[14: 1]	(0=disarmed, 1=armed)
history_link	[12:13]	Self-relative displacement to prior MSCW

RCW (Tag = 7)

rs	[44: 1]	Restart indicator (0=initial, 1=restart mode)
psi	[43: 3]	The PSI code stream pointer component
pwi	[40:13]	The PWI code stream pointer component
control_state	[27: 1]	CS Boolean (0=normal state, 1=control state)
tfff	[26: 1]	TFFF (true false flip flop)
lex_level	[25: 4]	The lex level at which the activation record runs
offf	[21: 1]	OFFF (overflow flip/flop)
extf	[20: 1]	EXTF (external sign flip/flop)
ASD_ref	[19:23]	ASD_number

The history_link field occurs in the MSCW; it contains a self-relative displacement down the stack to the next MSCW on the historical chain. The link is valid in any MSCW, but is not significant in a pseudo AR.

The block_exit bit indicates whether or not a service interrupt is to be generated when the activation record is deallocated. The enter operators always initialize this bit to 0 (disarmed). It can be set to 1 by software, in which case it must be reset to 0 by software before an Exit operation can deallocate this AR.

The stack linkage retains state necessary to reestablish and resume the procedure that invoked the current procedure. The RCW preserves for exit the same components provided by the PCW for procedure entry, specifying the code stream pointer, lex level, control state, and privileges. The rs indicator indicates whether the first operator in the resumed code stream should execute in restart or initial mode. The general Boolean accumulators are also preserved.

Top-of-Stack Linkage

When a processor is bound to a stack, its processor number is stored as an integer in the base word of the stack. The stack is said to be active; the processor is said to be "running in" the stack. When the stack is inactive (has no processor bound to it), the base word contains a TSCW, which points to the topmost activation record in the stack.

```
TSCW (Tag = 3)

  stack_num      [46:15]     The stack_number identifying the stack whose base
                             contains this TSCW
  stack_disp     [31:16]     Displacement to topmost MSCW
                 [15: 1]     Must be zero
  offset         [12:13]     Offset of top of stack from topmost MSCW
```

There is an extra activation record on an inactive stack, consisting of an entered MSCW and an RCW. The MSCW.history_link designates the topmost MSCW on the stack when it was (and will become) active; the RCW defines the code stream associated with the active process. The MSCW.env_link is zero. The TSCW designates this extra AR; its displacement points to the MSCW and its offset is 1, designating the RCW.

Protected Object Word (POW)

Four forms of Protected Object Words (POWs) are defined:

- a **soft POW** that is reserved for software use
- an **interlock POW** that is used to represent interlocks
- a **restart POW** that can be used as an extra argument for operators in restart mode
- a **hard POW** that is recognized by the processor for containing TCU event tokens

In each case, the purpose of the data type is to provide unique representations of information that is protected from access by the normal fetch and store operators.

```
Soft POW (Tag = A, [46: 1] = 0)

              [46: 1]  0: denotes software POW
              [45:47]  Reserved for software
```

```
Interlock POW (Tag = A, [47: 4] = 4)

              [47: 4]  Binary 0100: denotes interlock POW
              [43: 5]  Reserved for software
owner_id      [38:15]  Stack number of lock owner
lock_control  [23:21]  Reserved for software use
broken_bit    [ 2: 1]  1 implies state is Broken
lock_unc_bit  [ 1: 1]  1 implies state is Locked_Uncontended
not_free_bit  [ 0: 1]  1 implies state is not Free


Restart POW (Tag = A, [47: 4] = 5)

              [47: 4]  Binary 0101: denotes restart POW
              [43:44]  Reserved for hardware


Hard POW  (Tag = A, [47: 2] = 3)

              [47: 2]  Binary 11: denotes hard POW
              [45: 2]  Reserved for software
              [43: 4]  Binary 0011: denotes event
              [39: 7]  Reserved for software
              [32: 1]  If 1, the IOP will not cause event
              [31:32]  TCU event token
```

Tag Values

Throughout this document, tag values are specified in hexadecimal notation.

In most operations, the application and recognition of tags is automatic. Only the Read-Tag and Set-Tag operators (RTAG and STAG) elicit evaluation or assignment of tag values in programs; these are used very sparingly in user programs.

The following tag levels uniquely determine the data type. User programs can use STAG to construct words of these types, although many other operators generate, manipulate, and transform operands.

Tag	Type	
0	single precision operand	
2	double precision operand	
4	tag 4 word:	non-operand bit vector
6	tag 6 word:	non-operand bit vector, uninitialized single datum

The following tag values characterize, but do not necessarily uniquely determine, the type:

Tag	Type
1	SIRW
3	untouched CSD
5	untouched DD
7	PCW
D,F	IndexedDD

User programs construct SIRWs, PCWs, and IndexedDDs using appropriate reference-generation operators. They occasionally modify such words, but not their tag values. They construct untouched DDs using STAG. Compilers place images of untouched CSDs into user program code files; they are not constructed with STAG.

User programs occasionally use RTAG to determine the types of words that are parameters to procedures; the following assumptions are valid:

- Operands, tag-4, and tag-6 words have unique even tags
- SIRWs have tag 1
- IndexedDDs have odd tags other than 1

For example, if the type of a formal parameter to a procedure is a discriminated union of {single operand, SIRW, IndexedWordDD}, the tag of the parameter word can be used to discriminate the actual parameter type. Within the specified union, an even tag denotes an operand and an odd tag denotes a reference with 1 marking an SIRW and any other valid odd value an IndexedDD.

Data Structure Access

All the data structures defined in this manual are involved automatically in the action of some operator. Data items can also be constructed or manipulated by software using relatively primitive operators, such as fetch, bit-field operations, and store. For most data types, primitive access is restricted to operating system code at a low level of abstraction; only a few types are subject to primitive access by code in user programs. The following paragraphs outline the primitive operations that user programs can apply to data of various types. This text is by no means a rigorous specification of such code.

Only operands are subject to unrestricted manipulation. The use of tag 4 and tag 6 words is governed by convention between the compilers and the operating system (MCP).

Construction of descriptors is limited to untouched DDs in which field [47:2] contains zero and [45:46] contains an encoding according to compiler/MCP convention.

Manipulation of DDs is restricted to copy descriptors, which are fetched with the LOAD and indexing operators. Copy DDs may be stored, provided that a copy must not continue to exist after the segment to which it refers is gone. Copy descriptors can be stored by ordinary programs only into normal activation records, where they may readily be found if necessary. Modification of copy descriptors by primitive means is permitted; however, it is restricted to altering the element_size field, altering the char_index field, and setting the read_only bit.

SIRWs are constructed by Name Call operators. SIRWs may be fetched with LOAD; they may be stored by user program code, provided that a stored reference does not outlive the referent. Modification of SIRWs by primitive means is permitted and is restricted to setting the bit reserved for software and (under special circumstances) altering the offset field.

Operators are provided to modify element size in copy DDs and the access rights in copy DDs. These operators should be used in preference to primitive operations in programs intended to run on the MCP/AS (Extended) systems.

PCWs are built on the stack by the Make PCW operator. They can be fetched only with Load Transparent and can be stored only into the appropriate activation record.

Primitive code in user programs should not access any other data structures. Specific operators deal with interlocks. Stack linkage words and code segment descriptors need not be accessed directly. ASDs and ASRWs do not occur within the addressing environment of user programs.

Section 3
Interrupts

This section generally applies to all MCP/AS (Extended) systems, but many details are specific to the A16.

Interrupt Mechanism

An interrupt is an automatic invocation of a procedure; the mechanism is defined as the common action aINTE. (A related mechanism, called accidental entry, is defined in Section 4 of this manual as common action aACCE.) Exit from the interrupt procedure, when practical, resumes execution of the interrupted code stream.

Operator-dependent interrupts (ODIs) are invoked directly by the current operator to request an MCP service required by the operator or to report a programming or operator fault. Spontaneous interrupts (SIs) generally cannot be associated with a particular operator; some involve execution of the code stream and some do not.

When an operator may report several interrupts, there is generally no requirement that one interrupt take precedence other than that imposed by the function of the operator. When processing actions are functionally sequential, interrupts generated by an earlier action take precedence; otherwise, interrupts generated by any part of the operator may be reported. (For example, if some item B is meaningful only when item A is interpreted in a particular way, an error in that interpretation of A must take precedence over an error detected in B.) If a conditional action of the operator is not performed, interrupts that might have been generated by it are not required. (For example, if the interpretation of A is such that B is not significant, then any errors that might have been detected in B need not be reported.)

Superhalt

A superhalt condition exists when the processor cannot process an interrupt condition. In such cases, the processor halts in a state from which normal continuation is not possible.

The A16 processor detects three classes of superhalt conditions:

1. When the restricted operation aMVST encounters an interrupt condition during the interval that the processor has no valid stack environment

2. When RTOD (which communicates with the MIU) and the restricted operator XFNC encounter a hardware error in the midst of a message exchange
3. When an interrupt loop is detected

An interrupt loop is detected through use of the Interrupt_Count register. The register is incremented by one at the beginning of interrupt entry; if the register value is seven prior to being incremented, a superhalt condition exists. The only way to change the register, other than by increments, is to use the restricted operator ZIC to set it to zero.

Vectored Interrupt Conditions

An interrupt reports the occurrence of some set of circumstances called the *interrupt condition*. Each distinct interrupt condition generates a different interrupt reference, which can invoke a unique procedure for that condition. An interrupt reference is an SIRW designating a fixed location within a dedicated stack called the *interrupt vector*; the referent is called the *interrupt slot*. The unqualified word "interrupt" is often used to refer to the interrupt slot, the reference, or the interrupt condition that generates that reference. The interrupt reference is the head of the reference chain for an interrupt procedure; the interrupt slot can contain another SIRW or the PCW for the procedure.

Most interrupt conditions are assigned a single slot; these are called *Condition-Vectored Interrupts (CVIs)*. Several conditions are assigned a separate slot for each operator that can generate the interrupt; these are called *Operator-Vectored Interrupts (OVIs)*. The interrupt vector contains two activation records: one for CVIs and one for OVIs. Not all the slots are assigned in either AR.

Interrupt Formats

Two interrupt formats, or calling sequences, are used: Programmed Operator Interrupt (POI) and Restartable Operator Interrupt (ROI). Each format is named for a particular application to which it is particularly suited, although neither is restricted to a single application.

POI

POI is an interrupt format that facilitates operator emulation. The arguments to the operator are provided as parameters to the interrupt procedure, so they are readily accessed by the procedure and are automatically discarded by procedure exit.

Operator emulation can be required because a defined operator is not implemented (such as, some obsolete operators), because the operation requires software assistance (such as, LOK/UNLK), or because normal domain or range of operation is exceeded (such as, arithmetic underflow). In emulation cases the interrupt procedure is expected to return any required results of the operation; the code stream resumes with the next operator.

Some spontaneous interrupts use the degenerate form of POI format with no parameter.

ROI

ROI is an interrupt format suited to reporting service exceptions so that the operator can be repeated, for example in restart mode. Input arguments to the interrupted operator are pushed onto the stack below the interrupt AR; the interrupt procedure is provided two parameters: P1 and P2. The parameters are both single words. P1 is always zero for the A16; P2 typically designates to software the object that is erroneous or in need of service.

Code-Stream Designation and Resumption

The interrupt entry generates an RCW that designates the resumption point in the code stream.

Often, the RCW designates the operator whose execution generated the interrupt. The VARI syllable is designated in the interrupt RCW for a variant operator. Similarly, the enter-edit operator is designated when an edit operator is interrupted; for table edit, an updated table pointer designates the edit operator. (An initial table pointer is an IndexedDD; in the restart mode of TEED or TEEU, the updated table pointer is an IndexedDD.)

Resumption of the code stream is a key concern in interrupt definition; important resumption issues include:

1. If the code stream can be resumed
2. If so, the action required by the interrupt procedure
3. The operator that is designated by the interrupt RCW
4. If that operator is to be executed in initial or restart mode

If the code stream can be resumed from the interrupt procedure, the required exit operator is specified; otherwise, the code stream is said to be defunct and must be aborted. The following is an explanation of this notation:

Abort	Resumption is impossible and undefined. (It may be possible to resume the program at some other point in the code after appropriate adjustment of the process stack.)
EXIT	The interrupt procedure must exit, not return.
RETN	The interrupt procedure must return the appropriate result value for the emulated operator.
RTN2	The interrupt procedure must return two appropriate result values for the emulated operator.
REXI	The interrupt procedure must fetch the P1 parameter to the top of the stack and execute the REXI operator.

For resumable ODIs, the RCW may designate the operator that generated the interrupt, the next operator in the code stream, or the successor operator (not necessarily in sequence); these resumptions are called Repeat, Next, and Continue, respectively. Resumable SIs are considered to occur between operators in the code stream; the RCW designates the next operator to be executed and the resumption is called Continue.

Although "resumption" is a misnomer for defunct code streams, the term is used with the following notation:

Abort-Lost:	The RCW is undefined
Abort-Here:	The RCW designates the interrupted operator
Abort-Next:	The RCW designates the next operator

Interrupt Parameters

For ODIs in POI format, the interrupt parameters are the arguments to the interrupted operator. Each argument occupies two words on the stack; if the argument type is not double-precision, the second word is zero.

For interrupts in ROI format, the second parameter, P2, varies according to the specific interrupt type; it is often used to designate the object of concern or other information about the circumstances of the interrupt. P2 is specified for each interrupt case that uses the ROI format.

For several interrupts, P2 is specified to be a "proper reference" (either an SIRW or an IndexedDD) to the object of concern. The P2 parameter is a copy of the reference the processor was evaluating when an exception was detected; if that reference is an address couple (in an operator parameter or PCW skeleton), it is *stuffed* to yield an SIRW.

The value of P2 or of any POI parameter may be used in several ways:

1. It is used by operating system software in performing some service invoked by the interrupt. Such use can occur for interrupts that have parameter values specified and have resumptions other than Abort. (Such interrupts are flagged "S" or "s".)
2. It is used by operating system or diagnostic software in analyzing software and hardware failures for reporting, logging, or recovery. This use is characteristic of alarm interrupts. It sometimes occurs with enforcement interrupts; for example, some software distinguishes an *uninitialized datum* from other unacceptable items in an Invalid Target.
3. It is available (typically in diagnostic dumps) for users who are trouble-shooting a failure, although it is not analyzed programmatically. This use characterizes all interrupts that can report significant parameter values and are not instances of the first two cases.

Interrupt parameters are never unrelated to the circumstances of the interrupt. P2 is never an original DD or a POW.

Restart Configuration

The A16 uses restart mode in two types of situations:

1. Accidental entry in Value Call and almost all interrupts in Value Call: the extra state is the current reference.
2. Most interrupts during all data array operators, except for short-source operators (which are repeated from the beginning when interrupted at a page boundary): the extra state defines the processing already accomplished and is in a restart POW. (Generally, the other arguments are unmodified.)

The extra state is left immediately below the interrupt AR and rs is set; the operator in restart mode consumes the extra argument.

Also, when rs is true, certain operators have additional extra arguments in order to simplify their implementation:

1. SRCH has a length equal to one greater than the result of GINX on the domain argument.
2. EXPU has a dummy source argument.

Operator Subdivision

The A16 is unable to complete the longest feasible Data Array operations within a single Loop Timer interval. Therefore, it provides for the long Data Array operators to be divisible. If some stimulus (such as, External Interrupt or TCU move-stack request) is pending, the operator is interrupted, the stimulus serviced, and the operator resumed. Data Array operator subdivisions are the only instances in which some stimulus service occurs within an operator. Masked stimuli remain pending if the processor is in control state; control-state software must avoid excessively long array operations.

The word Data Array operators (vector, word transfer, and search operators) are divisible between iterations.

The character Data Array operators (not including short source, unpack, and skip operators) are divisible upon completion of each destination word (source1 for character sequence compare operators and source for scan operators).

Due to the exceptional source/destination overlap requirements for MFLT, that operator is not subdivided after FLTF is set.

Interrupt Definition

Interrupts are defined in five categories, which are presented in this section under separate headings for convenience rather than as correspondence to any grouping of interrupts slots or formats. The following are the types of interrupt definitions:

Arithmetic
Service
Enforcement
Alarm
External

Within this document, information on interrupt conditions is presented in specific notation as described here.

Key information about each interrupt condition is summarized in a line containing six items in three groups, as follows:

```
OVI:SUBT    POI(x,y)            Next      RETN(big(x-y))       ODI   E
CVI  ROI                        Repeat    REXI(P1)             ODI   S
CVI  POI( )                     Continue  EXIT                  SI   Se
```

The condition group (CVI or OVI) and format (ROI or POI) are specified at the left. For OVIs, which are vectored by opcode, the operators are identified. For POIs, the parameters are indicated, often as dummy identifiers that are clear from the operator definition; an ellipsis indicates generically that the arguments to the interrupted operator are provided. Empty parentheses indicate the degenerate POI format in which there are no interrupt parameters.

The resumption situation is specified in the middle of the line, using the notation defined previously. Return values are indicated for resumptions involving return. The notation is local to each interrupt definition and often involves the POI parameters.

At the right end, the interrupt is informally classified as operator dependent or spontaneous (ODI or SI) and as service or error.

Interrupts are categorized as service requests or error reports, with the following notation:

S:	Service
Se:	Service, sometimes treated as error
SE:	Service by definition although likely to be Error
E:	Error
Es:	Error although capable of being interpreted as Service
ES:	Error by definition although often used as Service

Arithmetic Interrupts

Section 4 of this manual defines the arithmetic operations and includes five exception conditions:

Divide by Zero
Exponent Overflow
Exponent Underflow
Precision Loss
Integer Overflow

In the vectored interrupt mechanism, each exception generates one of only three generic interrupts, each an OVI. In some cases the interrupt procedure must examine the operator arguments in order to establish which exception to report, if any. When the code stream is to be resumed, the interrupt procedure must return the appropriate operator result.

The processor is permitted to generate an arithmetic interrupt in certain situations for which the operation, as defined in Section 4, is not exceptional. These interrupts are called *spurious*. The interrupt procedure must emulate the operator and produce a proper result.

A spurious arithmetic interrupt is deemed a service interrupt; the interrupt procedure emulates the operator and resumes the code stream without reporting any exception to the rest of the software. Otherwise, an arithmetic interrupt is deemed an error report. The interrupt procedure determines the nature of the exception, if necessary, and takes appropriate software action for that exception. If the code stream is to be resumed, the procedure supplies the appropriate exception result for the operator.

Throughout this section, the word "type" refers to single- or double-precision operands. Each arithmetic interrupt procedure must implement the type rules defined in Section 4 for the interrupted operator.

Arithmetic Overflow

Arithmetic Overflow interrupts are generated by arithmetic operators when the result exponent is too large to represent and by the divide operators when the divisor is zero. This interrupt reports the Exponent Overflow and Divide by Zero exceptions.

```
OVI:ADD     POI(x,y)          Next   RETN(big(x+y))                ODI  E
OVI:SUBT    POI(x,y)          Next   RETN(big(x-y))                ODI  E
OVI:MULT    POI(x,y)          Next   RETN(big(x*y))                ODI  E
OVI:MULX    POI(x,y)          Next   RETN(bigdp(x*y))              ODI  E
OVI:DIVD    POI(x,y)          Next   RETN(zero(x,y) or big(x/y))   ODI  E
OVI:IDIV    POI(x,y)          Next   RETN(zero(x,y))               ODI  E
OVI:RDIV    POI(x,y)          Next   RETN(zero(x,y))               ODI  E
OVI:SNGL    POI(x)            Next   RETN(bigsp(x))                ODI  E
OVI:SNGT    POI(x)            Next   RETN(bigsp(x))                ODI  E
```

The interrupt procedures for the divide operators report Divide by Zero when the second parameter (y) is zero. The code stream can be resumed by returning zero of the appropriate type, as follows:

```
zero(x,y)     z AND y AND 0
```

In all other cases of Arithmetic Overflow interrupt, the interrupt procedures report Exponent Underflow. For operators that produce real values, the code stream can be resumed by returning the largest representable magnitude of the appropriate type and sign:

```
big(x+y), big(x-y)   biggest value with type x OR y and sign of x
big(x*y), big(x/y)   biggest value with type x OR y and sign of x*y
bigdp(x*y)           biggest dp value with sign of x*y
bigsp(x)             biggest sp value with sign of x
```

The "biggest value" is a single- or double-precision operand with all bits in the exponent and mantissa fields set to one and the exponent sign set to zero.

Arithmetic Underflow

Arithmetic Underflow interrupts are generated by arithmetic operators when the result exponent is too small to represent. This interrupt reports the Exponent Underflow and Precision Loss exceptions. It may also report spurious conditions for several operators.

The interrupt is generated when the final result of a computation has an exponent value that is negative and too large in magnitude to be contained in the exponent field.

```
OVI:ADD     POI(x,y)        Next   RETN(R(x+y))                  ODI   S
OVI:SUBT    POI(x,y)        Next   RETN(R(x-y))                  ODI   S
OVI:MULT    POI(x,y)        Next   RETN(R(x*y) or zero(x,y))     ODI  ES
OVI:MULX    POI(x,y)        Next   RETN(RD(x*y)) or zerodp)      ODI  ES
OVI:DIVD    POI(x,y)        Next   RETN(R(x/y) or zero(x,y))     ODI  ES
OVI:RDIV    POI(x,y)        Next   RETN(x MOD y)                 ODI   S
OVI:SNGL    POI(x)          Next   RETN(zerosp)                  ODI  E
OVI:SNGT    POI(x)          Next   RETN(zerosp)                  ODI  E
OVI:NORM    POI(x)          Next   RETN(zero(x))                 ODI  E
```

For ADD, SUBT, and RDIV, the interrupt is always spurious; the procedure must emulate the operation and return a representation, in the proper type, of the defined result.

The interrupt procedures for the other dyadic operators must emulate the operation and determine which of the following three actions to take:

1. If the result magnitude is less than half the smallest representable value, the procedure must report Exponent Underflow. The code stream can be resumed by returning zero of the proper type, such as:

   ```
   zero(x,y)          x AND y AND 0
   zerodp             dp zero
   ```

2. If the result can be represented in nonzero unnormalized form, but is too small to normalize, and has nonzero low-order bits that must be discarded because of the inability to normalize (R* < > R), the procedure must report Precision Loss. The code stream can be resumed by returning the nearest representation (in proper type).
3. If neither of the two previous situations holds true, the interrupt is spurious; the result can be represented in nonzero unnormalized form without loss of precision (R* = R). The procedure must resume the code stream by returning a representation, in the proper type, of the defined result; no exception is reported to the rest of the software.

For NORM, SNGL, and SNGT, normalization is required and therefore, the interrupt is never spurious. The interrupt procedures must report Exponent Underflow. The code stream can be resumed by returning a zero of the proper type:

```
zero(x)      zero of same type as x, for example x AND 0
zerosp       sp zero
```

Integer Overflow

An Integer Overflow interrupt is generated for the Integer Overflow exception.

The scale-left operators generate an Integer Overflow interrupt only if the argument to be scaled is out of range; if the scaled result is out of range, the operator sets OFFF and generates no interrupt.

```
OVI:NTGR,NTIA        POI(x)       Next   RETN(maxsp(x))         ODI  E
OVI:NTGD,NTTD        POI(x)       Next   RETN(maxdp(x))         ODI  E
OVI:IDIV             POI(x,y)     Next   RETN(max(x/y))         ODI  E
OVI:RDIV             POI(x,y)     Next   RETN(zero(x,y))        ODI  E
OVI:BCD              POI(x)       Next   RETN(nines(param))     ODI  E
OVI:DBCD             POI(x,n)     Next   RETN(nines(n))         ODI  E
OVI:SCLF             POI(x)       Next   RETN(maxdp(x))         ODI  E
OVI:DSLF             POI(x,s)     Next   RETN(maxdp(x))         ODI  E
OVI:SCRF             POI(x)       Next   RETN(nines)            ODI  E
OVI:DSRF             POI(x,s)     Next   RETN(nines)            ODI  E
OVI:SCRS             POI(x)       Next   RTN2(nines,maxdp(x))   ODI  E
OVI:DSRS             POI(x,s)     Next   RTN2(nines,maxdp(x))   ODI  E
OVI:SCRR,SCRT        POI(x)       Next   RETN(maxdp(x))         ODI  E
OVI:DSRR,DSRT        POI(x,s)     Next   RETN(maxdp(x))         ODI  E
```

All interrupt procedures for these interrupts report Integer Overflow. The code stream may be resumed by returning the appropriate values that are defined as follows:

```
max(x/y)     biggest integer with type x OR y and sign of x*y
maxdp(x)     biggest dp integer, (8**26)-1, with sign of x
maxsp(x)     biggest sp integer, (8**13)-1, with sign of x
nines        4"999999999999"
nines(n)     IF n <= 12 THEN nines ELSE DOUBLE(nines,nines)
zero(x,y)    zero with type of x,y, for example x AND y AND 0
```

Note that the type of the operand to be returned by the interrupt procedure for BCD is determined by the value of the code parameter. The procedure must fetch the syllable immediately preceding the one designated by the interrupt RCW.

Service Interrupts

The interrupts defined in this section usually constitute requests for an MCP service that is an extension of the hardware operators. In some situations, especially limiting cases, no service can be provided and the interrupt must be treated as an error.

Absent Segment

An Absent Segment interrupt is generated by operators requiring access to a segment that is not present in memory. The interrupt is generated when access is attempted via a touched DD or a PCW or RCW whose ASD_ref field designates an ASD with ASD1.present_to_proc equal to zero.

Two Absent Segment conditions are defined, one for code and one for data. Absent edit tables are reported as absent code.

Absent Code Segment

For Enter, Exit, and Dynamic Branch:

```
CVI ROI                          Continue  EXIT          ODI  S
```

The interrupt is generated when access is attempted via a PCW or RCW whose ASD_ref field designates an ASD with ASD1.present_to_proc = zero. P2 is identical to the interrupt RCW, which references the absent segment.

The interrupt RCW contains the code stream pointer whose attempted distribution encountered the absent segment; after the code segment has been made present and the ASD has been changed, an exit from the interrupt procedure will complete the enter, exit, or dynamic branch into the intended code segment. Note that RCW.rs is set in the interrupt RCW if an exit to restart mode was interrupted; this is an example of the preservation of restart mode.

For Table Edit:

```
CVI ROI                          Repeat    EXIT          ODI  S
```

The P2 parameter is an IndexedDD designating the edit table. The interrupt RCW designates the TEED or TEEU operator.

Absent Data Segment

```
CVI ROI                          Repeat    REXI(P1)      ODI  Se
```

- P2 must be an indexed DD that referenced the absent segment for those cases requiring service.
- On the A16, P2 can be an SIRW whose referenced segment is absent; however, such cases are treated as errors.

The service requested by an Absent Segment interrupt is for the software to make the absent segment present, after which the code stream can be resumed. It is the responsibility of software to ensure that all required segments are present for progress to be made. For example, various pointer operators require some or all of source, destination, and table (edit, translation, or membership) segments. Another example is that a short-source or unpack operator requires both pages be present if the operation crosses a page boundary, since the A16 repeats such operators from the beginning. Any required data segments and the code segment must be resident simultaneously for the operator to execute to completion.

Activate Object

```
CVI ROI                          Repeat    REXI(P1)        ODI  S
```

This interrupt is generated by the LPOW operator when the target word is an operand instead of a POW.

The P2 parameter is a proper reference to the inactive object. The expected service is to replace that object by a POW.

Binding Request

The Binding Request interrupt is not formally defined for the A16. However, the Binding Request interrupt must be provided in implementations that can run code files compiled with Mark 3.7 software. The interrupt is specified here, but does not appear in the appendices.

```
CVI ROI                          Repeat    REXI(P1)        ODI  S
```

The interrupt is generated (instead of Invalid Target) by the indexing, value-call, and normal-store operators when reference evaluation encounters a touched DD with element_size equal to seven; P2 is an SIRW that references the subject DD.

Block Exit

```
CVI ROI                          Repeat    EXIT            ODI  S
```

Exit operators generate a Block Exit interrupt when an attempt is made to deallocate an activation record that has MSCW.block_exit equal to one. P2 is zero.

Locking

```
OVI:LOK     POI(timeout,ref)     Next      RETN(result)    ODI  S
```

The Locking interrupt is generated by the LOK operator when the operating system service is required to resolve interlock contention. The first interrupt parameter is the timeout argument, the second is the reference argument.

The interrupt procedure is expected to perform the operation and return the appropriate result value for the LOK operator.

Missing PCW

```
CVI ROI                          Repeat    EXIT          ODI  SE
```

A missing PCW interrupt is generated by the Enter operation when the reference chain does not yield a PCW. P2 is an SIRW to the word that is not a valid SIRW or PCW. If the word at F+1 is not an SIRW or PCW, the Missing PCW interrupt is generated and P2 is an SIRW designating that word; in this case, SIRW.stack_disp designates the inactive MSCW at F.

If the offending word is a binding request token (such as, untouched DD), the operating system can perform the service of replacing that token by an appropriate reference. In other circumstances, the interrupt is reporting an error associated with the entry reference chain.

Stack Overflow

```
CVI POI( )                       Continue  EXIT          SI  Se
```

Stack Overflow indicates that a push onto the expression stack has caused the size of the stack to exceed its limit. The interrupt is a request to the MCP to extend the stack segment. The interrupt is classed as Spontaneous because it is generated when the object is pushed into memory, not necessarily when it is placed into registers at the virtual top of stack. As an SI, Stack Overflow is said to occur between operators. Any operation that induces stack overflow completes normally. The stack and code stream pointer are left in a configuration consistent with resuming the code stream with an enlarged stack segment.

The Stack Overflow condition is established when a word is pushed at a stack displacement greater than or equal to ASD2.AS_length for the stack. Once established, the condition is retained until an interrupt is generated or a Move-Stack operation is performed. The interrupt is generated when the condition is established and the interrupt is enabled, as soon as practical after the operator that induced the push of words into physical memory or the operator that enabled the interrupt.

Stack Overflow interrupt is disabled by its generation or by the successful initiation of a Move-Stack operation; it is enabled by execution of SPRR for "register" 36 or by successful completion of a Move-Stack operation. The interrupt is not masked by the CS Boolean.

While Stack Overflow is disabled, bounds checking on the current stack is disabled.

Touch

A Touch interrupt is used by operators to involve the MCP in the allocation or linkage of resources represented by untouched descriptors. The MCP service requested is to replace the untouched VSD with the appropriate item, which is typically an original VSD or a reference.

The P2 parameter is a proper reference to the untouched VSD.

Three Touch conditions are defined: Touch Reference, Touch Load, and Touch CSD.

Touch Reference

```
CVI  ROI                          Repeat    REXI(P1)   ODI  S
```

A Touch Reference interrupt is generated when an untouched DD is encountered as a reference target by any operator (except ENTR and EVAL) that evaluates reference chains. (ENTR generates Missing PCW interrupt; EVAL treats the untouched DD as a valid target.)

Touch Load

```
CVI  ROI                          Repeat    REXI(P1)   ODI  S
```

The Touch Load interrupt is generated when an untouched DD is encountered in a fetch attempt by LOAD or the final load step of NXLN.

Touch CSD

```
CVI  ROI                          Repeat    EXIT       ODI  S
```

The Touch CSD interrupt is generated when an untouched Code Segment Descriptor is encountered by the MPCW operator.

The initial reference is the PCW skeleton; P2 is an SIRW generated by considering the sdll and sdi fields as the lambda and delta components of the *unstuffed* address couple. (Note that in this case "delta" can be 13 bits wide.)

The Touch Reference interrupt can occur at a page boundary in a data array operator, as a descriptor is reindexed for the next iteration. In addition to the P2 parameter designating the untouched DD, the operator must preserve a consistent set of arguments below the interrupt AR. Depending on the situation, three techniques are used by the A16:

1. The pointer skips, though defined as array operators, are effectively indexing operators.
2. The short-source and unpack operators are repeated from the beginning.
3. Restart mode is invoked in all other cases, causing the descriptors to be reindexed at the beginning of the next iteration. This approach (or some equivalent) must be used for the general case of the vector operators, in which the reindex increment (stride) is not necessarily +1 and repetition of an iteration could produce incorrect results.

Unimplemented Operator

```
OVI:...              POI(...)  Next EXIT/RETN/RTN2      ODI  S
```

The Unimplemented Operator interrupt is generated whenever a primary or variant operator is encountered for which an operator is defined in this document, but not implemented (at least in part) in the processor.

The following actions are required in order to use operator emulation:

- The appropriate number of stack arguments must be present as interrupt parameters
- The interrupt RCW must be generated to point at the first syllable past the operator (including any code parameters)
- An interrupt procedure must have been provided in OVI slot 3 for the operator; the procedure can be tailored for that one operator. The procedure receives the operator arguments as parameters in POI format. Any required code parameters must be fetched by the procedure from the code segment. The procedure must emulate the operation and then exit, returning the specified operator results, if any.

Unlocking

```
OVI:UNLK    POI(ref)          Next      RETN(result)   ODI  S
```

The Unlocking interrupt is generated by the UNLK operator when operating system service is required to resolve interlock contention. The interrupt parameter is the reference argument.

The interrupt procedure is expected to perform the operation and return the appropriate result value for the UNLK operator.

Vector Arithmetic Exception

OVI	POI(POW)	Repeat	RETN(POW)	ODI	S
OVI	POI(scalar, POW)	Repeat	RTN2(scalar, POW)	ODI	S

This interrupt is reported by vector operators on the A16 when an arithmetic fault occurs during the execution of the operator. It is also used to report spurious interrupt conditions that occur in vector operators, but not in the equivalent scalar operators.

Vector operators that do not include a scalar stack item pass one parameter to the interrupt procedure in POI format. This parameter has a POW tag and contains the iteration count for the operator in bits 31:32. Vector operators that do include a scalar stack item pass two parameters in POI format. The first is the scalar argument (single or double precision) and the second is the iteration count.

If the interrupted code stream is to be resumed, system software needs to obtain the data items, perform the appropriate arithmetic operations on them, update the result location, increment the iteration count by one, and return the scalar (if any) and iteration count arguments. The resumption condition is Repeat-Return2 or Repeat-Return, depending on the presence or absence of a scalar stack item.

Actual arithmetic exceptions (excluding vector operator spurious conditions) cause the appropriate arithmetic operator to generate an arithmetic OVI. That procedure decides whether to report an error and whether to return an emulation result to the vector arithmetic interrupt procedure.

Enforcement Interrupts

The interrupts in this group are generated primarily to enforce the preconditions and invariants defined for proper operator function. (In some cases, the MCP may take corrective measures or otherwise remove the error situation and resume the code stream, in which case the interrupt was effectively a request for service. Such action is possible only with non-Abort resumption situations.)

There are some differences among MCP/AS (Extended) systems regarding the extent of enforcement; details regarding interrupt format vary widely. The A16 responds as described in the following paragraphs.

Bounds Error

A Bounds Error is generated for any of the following reasons:

1. The index argument to an indexing or OCRX operator cannot be integerized. For indexing operators, P2 is zero; OCRX is reported as an OVI.
2. The index argument to an OCRX operator is less than one or exceeds the ICW_limit value in the index control word. P2 is zero.
3. An attempt is made to index an unindexed DD or reindex or update an IndexedDD so that the new virtual index is less than zero or exceeds the length of the virtual segment. P2 is the out-of-bounds IndexedDD, with an unsigned index.

4. The word index of a DD exceeds field capacity: 2**20-1 for a WordDD, 2**16-1 for a CharDD, or 2**13-1 for an edit table designator. P2 is that IndexedDD with a truncated index, unless the word index magnitude exceeds 2**32-1, in which case P2 is zero.

5. An attempt is made to access an item via an IndexedDD whose word index is greater than or equal to the word length of the actual segment designated by its ASD_ref. For an IndexedDoubleDD, both referent words must be in-bounds. P2 is the out-of-bounds IndexedDD.

   The final load step of NXLV and NXLN includes this check; those operators are defined as developing and then consuming an IndexedWordDD.

   Accesses to membership and translate tables are subject to this check.

   Accesses to source and destination sequences in pointer operators are subject to this check, except for the cases defined to generate Destination and Source Boundary interrupts.

6. The offset into an actual segment exceeds ASD2.AS_length-1, for:

   a. accesses via SIRWs, entered MSCWs, or TSCWs: P2 is the reference in SIRW format.

   b. accesses to code, whether sequential, as static or dynamic branch destinations, or via procedure entry/return: P2 is an unpaged IndexedDD that references the out-of-bounds code word.

   c. accesses relative to register values (such as F, D[LL]). However, pushing a word onto the current stack beyond its end constitutes a Stack Overflow condition; the store must be performed and Bounds Error interrupt cannot be generated. P2 is an SIRW to the current stack whose stack_disp field equals the register value.

When Stack Overflow checking is disabled, no Bounds Error interrupt is generated for any access to the stack segment. Software must not use a DD to access the stack beyond its defined AS_length.

Resumption is defined for the indexing and OCRX operators.

For the case of OCRX with an index argument that cannot be integerized:

```
OVI:OCRX   POI(TOS2, TOS)      Next      RETN(result)   ODI  Es
```

For the other cases of OCRX and all cases of indexing operators:

```
CVI ROI                        Repeat    EXIT           ODI  Es
```

In all other cases, the code stream need not be resumable:

```
CVI ROI                        Abort-Here             ODI  E
```

Destination Boundary

```
CVI ROI                          Repeat    REXI(P1)       ODI  ES
```

A Destination Boundary interrupt is generated by pointer operators (other than TWOD) that attempt to transfer data to the first element beyond the end of the destination virtual segment.

P2 is the destination descriptor updated to the last+1 element of the segment.

This interrupt ordinarily reports an error, however, there are two possible services that software can perform:

1. It can lengthen the destination segment.
2. It can terminate the operation, discarding the rest of the intended transfer.

Software may extend the destination segment in one of several ways:

1. An unpaged segment can be extended by providing a larger actual segment associated with the same ASD.
2. A paged segment with a short last page can be extended by providing a larger actual segment associated with the same last-page ASD.
3. A paged segment with a complete last page can be extended by providing a larger page table associated with the same ASD.
4. An unpaged segment can be converted to a paged segment. The ASD associated with the original segment can be used either as the new page table or as a page; an additional ASD is needed for the other.

In the first three cases, the code stream can be resumed without any modification of the stack arguments. The fourth situation requires that all touched DDs referencing the destination segment (including the destination argument on the stack below the interrupt AR) be modified to designate the new segment; the tags must be changed and the ASD_ref and index fields may require change.

Unconditional transfer operators (TUND and TUNU, TRNS, TWSD and TWSU) may be terminated by setting the length argument (on the stack below the interrupt AR) to zero prior to resuming the code stream. The resumed operator will terminate. These operators always place their arguments on the stack in their original relative positions; the length is the topmost operand below the interrupt AR. Note that there is a restart POW above the arguments.

False Assertion

```
OVI:ASRT,AVER        POI(b)    Next      EXIT           ODI  ES
```

The False Assertion interrupt is generated by the ASRT and AVER operators when the stack argument is False.

The interrupt is an error indication on the part of the program invoking ASRT or AVER, but it is also a service request for message generation or other software-defined error handling. The Boolean argument is passed as the interrupt procedure parameter. The code parameter to the ASRT operator can be fetched from the code segment; the interrupt RCW points at the syllable following the parameter.

Invalid Access

```
CVI ROI                          Abort-Here              ODI  E
```

An Invalid Access interrupt is generated when an operator attempts to write to memory via a DD with write access disabled. If the DD is a stack argument, P2 is zero; otherwise, P2 is the SIRW that referenced the DD.

Pointer operators that "require a destination" may generate the interrupt when the destination argument is read-only, whether or not a write is attempted.

Invalid Argument or Parameter

```
CVI ROI                          Abort-Here              ODI  E
```

This interrupt is a notice that a stack argument or code parameter for an operator does not meet the type and value-range constraints that are preconditions for the operator. Most of the preconditions cited in the operator definitions in this manual are enforced by checks that generate this interrupt when not satisfied. The bulk of the checks are on stack arguments, so the interrupt is often referred to as Invalid Argument. P2 is zero for all cases.

Invalid Argument or Parameter tests fall into the following categories:

- **Type checks:** Data types are defined by tags and, in some cases, additional discriminator bits within the word. Some data types are defined partly by context, so a tag check provides a partial verification. An Uninitialized Argument interrupt is generated by the A16 instead of Invalid Argument whenever the offending argument has a tag of 6 or 4.
- **Reindex checks:** Indexing, Data Array, and set-element_size operators require indexed-descriptor arguments to have reindex enabled; the reindex-enable bit may be considered part of the data-type definition for acceptable arguments. (Indexing operators generate this interrupt only for the Descriptor Indication argument; they generate an Invalid Target interrupt if a reindex-disabled indexed DD is encountered as a referent.)
- **Argument value checks:** An integer operand or a field in a word of some other type may be constrained to a particular range or subset of values.
- **Code parameter value checks:** A code parameter may be constrained to a particular range or subset of values.
- **Address couple checks:** the lambda (lex level) component must not exceed LL (the lex level at which the processor is running)

- **Consistency checks:** some operators place type or value restrictions on one argument, depending on some characteristic or another. Examples include element_size matching in pointer operators and type consistency between store objects and targets.
- **Integer magnitude checks:** Non-Integer Argument exception arises when any operator except NTGR invokes the RaI (algebraic round to integer) function and the result is TooBig. If the argument to be integerized is used as an Index, the exception is reported as a Bounds Error; otherwise, it is reported as an Invalid Argument. (Note that often an argument range is limited to a subset of the integers; in such cases, the more limiting check is specified. The TooBig result from the RaI function is regarded as greater than any representable value, even when the argument of RaI is negative.)

Invalid Code Word

An Invalid Code Word interrupt indicates that a word accessed from the current code segment is not a Program Code Word. It is generated in table edit mode if the tag of the word is not zero and in all other modes of the tag is not three.

The interrupt is classified as an SI because it is not associated with an identifiable operator. However, it is not at all independent of the code stream.

- Primary or Variant:

```
CVI ROI                          Abort-Here                SI  E
```

  If the processor is parsing code other than in edit mode, the interrupt RCW designates the first syllable of the operator that contains a syllable in the invalid word. If a VARI prefix has been encountered, that syllable is designated. (If a branch, enter, or exit destination begins in the bad word, the RCW designates that destination.)

- Edit:

```
CVI ROI                          Abort-Here            SI  E
```

  When the processor is parsing code in edit mode, the interrupt RCW designates the enter-edit operator; the arguments to that operator are below the interrupt AR. For table edit, an updated table pointer designates the first syllable of the operator that contains a syllable in the invalid word.

For all cases on the A16, P2 is an unpaged IndexedDD that references the offending code word.

Invalid Operator

```
CVI POI( )                     Repeat    EXIT           ODI  Es
```

An Invalid Operator interrupt is unconditionally generated by execution of NVLD (Invalid Operator). No other operator generates this interrupt.

Invalid Target

```
CVI ROI                          Repeat    REXI(P1)       ODI  Es
```

An Invalid Target interrupt indicates that evaluation of a reference produced an unexpected result. It is generated by operators (other than Enter) that evaluate references and reference chains if evaluation of an address couple parameter, SIRW, or IndexedWordDD produces an item that is neither a valid reference in the chain nor a valid target item that terminates the chain. The definitions of valid reference chains and valid target items vary according to the function of the operator.

The indexing operators generate the interrupt if an indexed DD with reindex disabled is encountered as the target of an SIRW; the reindex-enable bit may be considered part of the data-type definition for acceptable targets. (If the Descriptor Indication argument is an indexed DD with reindex disabled, an Invalid Argument interrupt is generated.)

The interrupt is generated by vector and pointer operators that encounter unacceptable words in a vector, (source or destination) sequence, or (membership or translate) table.

The P2 parameter is a proper reference to the offending target word. For a pointer operator, P2 may be an IndexedCharDD.

Invalid Target can be considered a generic condition for the discovery of an unexpected word during reference evaluation. In many special cases, some other interrupt is generated instead. In particular, the Enter operators generate Missing PCW interrupts and some other operators generate a Touch interrupt if the target is an untouched DD.

Pack Unindexed DD

```
CVI ROI                          Repeat    EXIT           ODI  Es
```

The Pack Unindexed DD interrupt is generated (on the A16 only) by the pack and input convert operators when the source is an unpaged unindexed copy DD. The operator may be repeated if the MCP replaces the source by the same DD indexed by zero. P2 is zero.

Source Boundary

```
CVI ROI                          Repeat    REXI(P1)       ODI  Es
```

A Source Boundary interrupt is generated by TRNS, TUND, TUNU, TWSD, and TWSU operators that attempt to transfer data from the first element beyond the end of the source virtual segment. Other pointer operators that transfer characters from a source segment to a destination also generate this interrupt under the same circumstances.

P2 is the source descriptor updated to the last+1 element of the segment.

A Source Boundary interrupt usually reports an error, although under some circumstances the software may perform a service by supplying the rest of the missing source or terminating the transfer.

To cause the operation to continue with an operand source, the software must replace the source pointer argument on the stack below the interrupt AR by an operand. (It must also set P1.[38:1]:=1 prior to returning P1 with REXI, for those A Series systems that interpret REXI as RETN.) For unconditional-transfer operators (TUND and TUNU, TRNS, TWSD and TWSU), the source-pointer argument is immediately below the length argument, which is the topmost operand below the interrupt AR. Note that there is a restart POW above the arguments.

The operation may be terminated as described under "Destination Boundary" earlier in this section.

Stack Underflow

CVI POI( )	Abort-Here	SI	E

Stack Underflow indicates that an operator attempted to pop an argument from an empty expression stack. The expression stack is the set of locations whose addresses are in the range {MAX(F+1,D[LL]+2) to S}; the A16 enforces the lower bound as F+1 singly. A Stack Underflow interrupt is generated when, in order to provide an argument to an operator, it is necessary to pop a word from the stack, but S is less than the lower bound.

This interrupt is classified as an SI (even though it always results from a specific operator attempting to consume an argument from an empty expression stack), because no parameters are passed.

Structure Error

CVI ROI	Abort-Here or Abort-Next	ODI	E

A Structure Error interrupt is generated when the processor discovers an inconsistency in one or more of the critical data structures that the processor requires to execute operators: a stack, the ASD table, or a page table. The interrupt usually indicates a serious failure of hardware or system software. Circumstances that result in interrupt generation are listed separately for each of the three types of structures, which are the following:

- Stack Structure Errors
- ASD Structure Errors
- Page Structure Errors

Stack Structure Errors

The following notation is used:

Mem[a]	= contents of memory word at address a
Stk[d]	= contents of current stack at displacement d
HistLink	= displacement computed from an MSCW history link
EnvLink	= address computed from an environment link
.ASD_num	= .stack_num or .seg_num for normal or pseudo link

The Enter, Exit, aMVST, and aEVLK operations generate Structure Error interrupts under any of the following conditions:

```
aEVLK:  Mem[EnvLink] not entered MSCW
aMVST:  Mem[ASD1[destination stack_number].address] not TSCW
Enter:  Stk[F] not inactive MSCW
        Stk[F].history_link_I >= 2**13
Exit:   Stk[D[LL]] not entered MSCW
        Stk[HistLink].history_link_I >= 2**13
        Stk[HistLink] not {inactive MSCW, entered MSCW}
        Stk[D[LL]+1] not RCW
```

Checking stack linkage words (MSCW, RCW, TSCW) only occurs whenever the word is accessed. Operators that update display registers may traverse part, but not necessarily all, of the environmental chain for the new environment. If the word is not the correct type, P2 is nearly always an SIRW that references it.

The checks for history_link_I >= 2**13 detect a situation called Stack Frame Overflow, which is the only Structure Error case that can be caused solely by a user programming error. P2 is zero.

ASD Structure Errors

An ASD structure error is detected in the course of evaluating an ASD_number, which may be derived in any of several ways:

1. Accessing data via a descriptor: the ASD_number is from the ASD_ref field of the DD.
2. Accessing code via a PCW or RCW: the ASD_number is from the ASD_ref field of the control word.
3. Evaluating an Environment Link: the ASD_number is from the env_link field of an SIRW or MSCW.
4. Accessing the current stack: the ASD_number is derived from the SNR register.
5. Accessing the current code segment: the ASD_number is in the PSN register.

In all of these contexts, the following violations generate Structure Error interrupts:

```
ASD1[ASD_number].tag < > 8: P2 is the absolute address of ASD1.
ASD2[ASD_number].tag < > 8: P2 is the absolute address of ASD2.
ASD1[ASD_number].address MOD 2**AW + offset >= 2**AW: P2 is the reference
                         whose evaluation caused the overflow.
```

Page Structure Errors

Violations of the invariants of paged array structure generate Structure Error interrupts:

- ASD3 designated by indexed paged DD not unpaged IndexedDD: P2 is the absolute address of ASD3.
- page-table word accessed by indexing unindexed paged DD or by reindexing ASD3 not {unpaged unindexed DD, untouched DD}: P2 is an unpaged IndexedDD that references the page-table word.

Undefined Operator

An Undefined Operator interrupt is generated for the attempted execution of an operator whose encoding is not valid in the context.

Primary and variant encodings are expected in the normal succession of operators in the code stream, whether accessed sequentially or subsequent to branch or subroutine operators. Edit encodings are expected in the code stream following an enter single-edit operator or in a table designated by an enter table-edit operator. The ENDE, ENDF, and INSC operators are undefined following EXPU, EXSD, or EXSU; edit operators requiring a source are undefined following EXPU.

Ordinarily, the resumption is Repeat EXIT (although useful resumption of the code stream is unlikely):

```
CVI POI( )                    Repeat  EXIT            ODI  E
```

If an edit operator was expected, the resumption is Abort-Here. The RCW points to the enter-edit operator and, for table edit, the table pointer is updated to point to the offending code syllable:

```
CVI POI( )                    Abort-Here              ODI  E
```

Uninitialized Argument

```
CVI ROI                       Abort-Here              ODI  E
```

This interrupt is a special case of Invalid Argument; it is generated when an argument has a tag of 4 or 6 and fails to meet the type criteria for that operator. P2 is zero.

Systems that pass an argument with a tag of 4 or 6 as an interrupt parameter can report this condition as Invalid Argument.

Alarm Interrupts

Alarm interrupts are triggered by hardware fault detection. The interrupt RCW typically points to the operator that was executing when the fault was detected; resumption is often not possible. Most discussion applies only to the A16.

Hardware Error

This interrupt indicates that the processor detected an error in its own operation.

Two Hardware Error conditions are defined, one for situations in which resumption is possible and one for which it is not:

Hardware Error - Here

```
CVI ROI                    Repeat  EXIT              SI  E
```

Hardware Error - Lost

```
CVI ROI                    Abort-Lost                SI  E
```

The P2 parameter for both repeat and abort hardware errors is formatted as follows:

```
[35:4]  PCU PI
            1  Bad PTM State
            8  PTM (micro-op) Parity
            9  PTM (literal) Parity
            A  Control Store Parity
            B  PTLL (LL) Parity
            C  ACAM Association Parity
            D  ACAM Duplicate Hit
            E  Unused
            F  Redo TruncCodeInx Parity

[28:5]  DCU PI
           14  Undefined Encode
           15  ASD_ref Residue & Misc. Field Parity
           16  Word Index Residue
           17  Misc. Control (on DARF & DCAL chips)
           18  Control Store Parity
           19  Op Q Parity (TruncCodeInx & ACAMptr)
           1A  Op Q Parity (rest of Q)
           1B  TOS Assignment List (RAL) Parity
           1C  Bad DLM state
           1D  Unused
           1E  Bad Tag from EU
           1F  Pre-fetch Failure
```

```
[23:8]   Interrupt Syllable PI

   [23:4]  Word Type
            0  Target
            2  ASD1
            3  ASD2
            4  Page DD
            6  PT ASD1
            7  PT ASD2

   [19:4]  Event
            8  ACU Bounds Residue
            9  CCU Address Residue
            A  CCU Address Parity
            B  ACU Hit Residue
            C  MAU Hardware Error
            E  VU Hardware Error
            F  ACU Command Error

[15:4]  General CPM PI

   [15:3]  Unused
   [12:1]  Bad Repeat

[11:4]   EU PI
            8  ERT Parity
            9  EIS Parity
            A  EWD Residue
            B  EWD Residue
            C  Code Index Parity
            D  TruncCodeInx Parity
            E  Control Store Parity
            F  Operator Queue Parity

[ 7:8]   Asynchronous PI

   [ 0:1]  ACAM Deallocation Parity
   [ 1:1]  Cache Data Parity
   [ 2:1]  CCU Replace Data Queue Parity
   [ 3:1]  CCU Address Recoverable
   [ 4:1]  CCU Control
   [ 5:1]  CCU Address Non-recoverable
   [ 6:2]  MAU Encode
            1  MAU Internal
            2  Interface Nonfatal
            3  Interface Fatal
```

Only one of the previous types of errors is reported in any P2; for example, only one of the PCU PI, DCU PI, Interrupt Syllable PI, EU PI, and Asynchronous PI fields is non-zero. It is possible for the whole P2 word to be zero if Error Reporting Chip itself detects a problem. This case should be reported as a "Bad ERT" state.

Invalid Address

This interrupt indicates an attempt to address a word of memory that does not exist on the system. P2 is the absolute address.

Two Invalid Address conditions are defined, one for situations in which the attempted access can be associated with a specific operator and one for which it cannot.

Invalid Address - Here

```
CVI ROI                    Abort-Here                SI  E
```

Invalid Address - Lost

```
CVI ROI                    Abort-Lost                SI   E
```

Loop Timer

This interrupt indicates an effectively infinite loop by an operator. It is triggered by expiration of a timer whose interval is sufficient for valid execution of any indivisible sequence of operators or one division of a divisible iterative operator.

A Loop Timer interrupt indicates a processor fault, except that a program can induce the interrupt by attempting to evaluate an SIRW chain that loops or to traverse a historical chain (exit) with a history link equal to zero.

The following operations are not subject to Loop Timer interrupt:

HALT (when the Halt Boolean is true)
STOP
Idle state (waiting for TCU message)
XFNC/RTOD (waiting for acknowledge)

Two Loop Timer conditions are defined, one for situations in which resumption is possible and one for which it is not:

Loop Timer - Here

```
CVI ROI                    Repeat  EXIT              SI  E
```

Loop Timer - Lost

```
CVI ROI                    Abort-Lost                SI  E
```

For both situations, P2 is zero.

Memory Data Error

```
CVI ROI                    Abort-Here              SI  E
```

This interrupt reports an attempt to read from memory a word that has an uncorrectable memory error. (No error may be reported regarding the prior contents of a memory word being overwritten by OVRD/OVRN or TWOD.) Single-bit read data errors are corrected by hardware and will not be reported by an interrupt.

P2 contains the absolute address in field [31:32]; field [47:16] is zero.

MSM Hung

```
CVI ROI                    Abort-Here              SI  E
```

This interrupt reports an attempt to address a word that the memory is unable to send to the processor.

P2 contains the absolute address in field [31:32]; field [47:16] is zero.

Message Error

```
CVI ROI                    Abort-Here              SI  E
```

This interrupt reports the reception of an erroneous message: either an acknowledge when the processor is not waiting for one or an unsolicited message when the processor already has one in its buffer that it has yet to read. P2 is zero.

External Interrupts

External interrupts are generated to report occurrences outside a processor, such as communication from another component of the system. The following are used by the A16.

Environmental Fault

```
CVI POI( )                 Continue EXIT           SI  E
```

This external interrupt is generated when the domain containing the processor is going to power off within five seconds. This interrupt is masked by CS.

Service Request

```
CVI ROI                    Continue EXIT           SI   S
```

This external interrupt is generated by the Task Control Unit or IO Unit. P2 is the last word of the four words in the message from that unit. The message can only be examined and, therefore, cause a Service Request when CS is false, unless the message is an acknowledge caused by XFNC.

Interrupt Vector

An interrupt is generated by invoking the aINTE action with an interrupt reference (SIRW) constructed by the processor. The components of the SIRW are:

```
stack_num       = 3
stack_disp      = 1 for CVI, 50 for OVI
pseudolink      = 0
offset          = value defined for particular interrupt
```

Each CVI has a single fixed offset.

Each OVI has a different offset for each operator:

```
primary:  opcode*4 - 511 + OVI_number
variant:  opcode*4 + 513 + OVI_number
```

An OVI_number is assigned generically for each OVI condition:

```
0:  False Assertion
0:  Locking
0:  Unlocking
0:  Arithmetic Overflow
1:  Arithmetic Underflow
2:  Integer Overflow
2:  Vector Arithmetic Exception
3:  Unimplemented Operator
```

The following lists define the offsets for all interrupts except Unimplemented Operator, which is always OVI slot 3 for the operator to be trapped. Offset values are shown in decimal and (hex). Unnamed offsets are unused.

CVIs: stack_disp =1

```
 1  (001)    Touch CSD
 2  (002)    Missing PCW
 3  (003)    Absent Data Segment
 4  (004)    Absent Code Segment
 5  (005)    Touch Reference
 6  (006)    Touch Load
 7  (007)
 8  (008)    Destination Boundary
 9  (009)    Source Boundary
10  (00A)    Binding Request (reserved)
11  (00B)    Block Exit
12  (00C)    Activate Object
```

```
13  (00D)
14  (00E)    Invalid Target
15  (00F)    Invalid Address - Here
16  (010)    Memory Data Error
17  (011)    Invalid Argument or Parameter
18  (012)    Bounds Error
19  (013)    Service Request
20  (014)    Structure Error
21  (015)    Invalid Operator
22  (016)    Invalid Access
23  (017)    Undefined Operator
24  (018)    Uninitialized Argument
25  (019)
26  (01A)
27  (01B)
28  (01C)
29  (01D)
30  (01E)
31  (01F)
32  (020)    Invalid Address - Lost
33  (021)    Stack Overflow
34  (022)    Stack Underflow
35  (023)    Invalid Code Word
36  (024)    Loop Timer - Lost
37  (025)    Hardware Error - Lost
38  (026)    Environmental Fault
39  (027)    MSM Hung
40  (028)
41  (029)    Loop Timer - Here
42  (02A)    Hardware Error - Here
43  (02B)    Message Error
44  (02C)
45  (02D)
46  (02E)
47  (02F)    Pack Unindexed DD
48  (030)
```

OVIs: stack_disp = 50 (0032)

```
 1  (001)    ADD     Arithmetic Overflow
 2  (002)            Arithmetic Underflow
 5  (005)    SUBT    Arithmetic Overflow
 6  (006)            Arithmetic Underflow
 9  (009)    MULT    Arithmetic Overflow
10  (00A)            Arithmetic Underflow
13  (00D)    DIVD    Arithmetic Overflow
14  (00E)            Arithmetic Underflow
17  (011)    IDIV    Arithmetic Overflow
19  (013)            Integer Overflow
```

21	(015)	RDIV	Arithmetic Overflow
22	(016)		Arithmetic Underflow
23	(017)		Integer Overflow
27	(01B)	NTIA	Integer Overflow
31	(01F)	NTGR	Integer Overflow
61	(03D)	MULX	Arithmetic Overflow
62	(03E)		Arithmetic Underflow
259	(103)	SCLF	Integer Overflow
263	(107)	DSLF	Integer Overflow
267	(10B)	SCRT	Integer Overflow
271	(10F)	DSRT	Integer Overflow
275	(113)	SCRS	Integer Overflow
279	(117)	DSRS	Integer Overflow
283	(11B)	SCRF	Integer Overflow
287	(11F)	DSRF	Integer Overflow
291	(123)	SCRR	Integer Overflow
295	(127)	DSRR	Integer Overflow
305	(131)	SNGT	Arithmetic Overflow
306	(132)		Arithmetic Underflow
309	(135)	SNGL	Arithmetic Overflow
310	(136)		Arithmetic Underflow
991	(3DF)	BCD	Integer Overflow
1023	(3FF)	DBCD	Integer Overflow
1025	(401)	ASRT	False Assertion
1047	(417)	OCRX	Bounds Error
1051	(41B)	NTTD	Integer Overflow
1055	(41F)	NTGD	Integer Overflow
1082	(43A)	NORM	Arithmetic Overflow
1085	(43D)	AVER	False Assertion
1217	(4C1)	LOK	Locking
1225	(4C9)	UNLK	Unlocking
1291	(50B)	SUM	Vector Arithmetic Exception
1295	(50F)	SUMA	Vector Arithmetic Exception
1299	(513)	DOT	Vector Arithmetic Exception
1303	(517)	DOTX	Vector Arithmetic Exception
1307	(51B)	SEQ	Vector Arithmetic Exception
1311	(51F)	POLY	Vector Arithmetic Exception
1423	(58F)	CPVS	Vector Arithmetic Exception
1431	(597)	VPV	Vector Arithmetic Exception
1435	(59B)	VMV	Vector Arithmetic Exception
1439	(59F)	VMVN	Vector Arithmetic Exception
1443	(5A3)	VTV	Vector Arithmetic Exception
1447	(5A7)	VOV	Vector Arithmetic Exception
1451	(5AB)	VUV	Vector Arithmetic Exception
1455	(5AF)	VPS	Vector Arithmetic Exception
1459	(5B3)	VTS	Vector Arithmetic Exception
1463	(5B7)	VPVS	Vector Arithmetic Exception
1467	(5BB)	VSPV	Vector Arithmetic Exception

Section 4
Operator Set

Except where specific references are made to the A16, this section applies to all MCP/AS (Extended) systems. Most operators are available in all A Series systems; see Appendix A to identify those operators introduced in MCP/AS (Extended) systems.

General Information

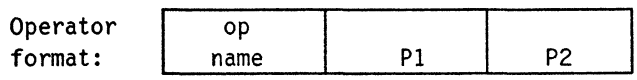
Operators and Code Streams

An operator is composed of an op code and up to four parameters. Op codes are typically one syllable and parameters, if any, are in the syllables following the op code. Op code and parameter mapping into syllables varies; operator formats are explicitly specified in this section and in Appendix A. (The term *parameter* is used in operator descriptions to describe items from the code stream; the term *argument* is used for items from the stack.)

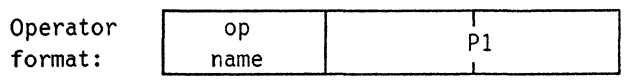
A code stream is considered to be a sequence of syllables fetched without regard to word boundaries, except for the special case of LT48 and MPCW operators.

In diagrams specifying op code and parameter interpretation, the operator name is used to represent its op code value. (Op code values are specified in Appendices A and C.) Vertical bars (|) denote syllable boundaries and dotted vertical lines (:) denote parameter boundaries not corresponding to syllable boundaries.

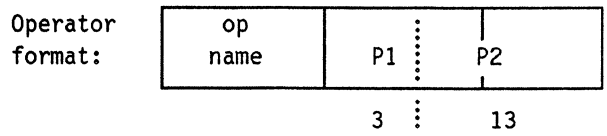
The following diagram shows a three-syllable operator, including two single syllable parameters.



The next diagram shows a three syllable operator, two syllables of which are a single parameter:



The final diagram shows a three syllable operator including two parameters that are mapped into two syllables. P1 is three bits and P2 is 13 bits.



Primary, Variant, and Edit Operators

Depending on context, there may be several interpretations of an op code syllable:

- Primary op codes are represented in a single syllable.
- Variant op codes are represented by two syllables of which the first is the primary operator VARI.
- Edit op codes are found in special tables (specified as an argument to an Enter Table Edit operator, TEED or TEEU) or in the code stream immediately following an Enter Single Edit operator (EXSD, EXSU, or EXPU).

Common Actions

The concept of a *common action* is used in this document for a function that is common to several operators. Common actions are defined to effect economy by reducing repetition, to improve precision, and to provide a convenient reference for citation. Common actions are given three- and four-letter names like operators with a prefix of *a*, as in *aACCE*. It should be emphasized that a common action specification is a rhetorical device used to specify operators; it does not define an operator.

Order of Execution

Operators from the code stream are executed in sequence until the code stream is redirected by a branch, enter, or exit operation. The processor can perform individual operations in parallel and execute operators out of order, as long as the effective order of the program is preserved. For example, any value fetched or consumed by an operator must be the value stored or produced by the closest predecessor operator affecting that memory or stack location. These store-before-read and produce-before-consume restrictions apply within the code stream on a single processor, not necessarily between code streams on separate processors. Two processors cannot share the same stack at the same time; therefore, produce-consume relationships cannot apply. To enforce a store-read sequence between processors, the processes must explicitly use synchronizing operations.

There are exceptions to the store-read sequence requirement when the words being read can have been captured by the processor as hidden optimization state; see "Program Restrictions Due to Hidden State" at the end of this section.

Interrupts and Resumption

When an operator cannot be performed or completed according to the definition of its normal operation, an interrupt is generated, the operator is aborted, and a procedure of the operating system is entered. The interrupt procedure is provided enough information to determine the circumstances and take the appropriate action.

Operator interrupts can generally be characterized as error reports or service requests, although the distinction is blurred. *Errors* can sometimes be corrected or used to induce special actions and *services* can be limited or denied.

Service requests are implicit in most operations that can encounter these situations; only occasionally are they explicitly pointed out in this document.

When a service request is honored (for example, a segment is made present in memory upon demand) or an error is accommodated, the code stream is resumed as the interrupt procedure is exited. Depending upon the circumstances, the interrupted operator is repeated or its successor proceeds. If the service cannot be performed or the error is unrecoverable, the interrupted code stream cannot be resumed. These resumption characteristics vary with interrupt type and circumstances.

For more information on interrupts, see Section 3.

Preconditions and Invariants

Throughout this manual, restrictions are defined on structures, argument types, values, and so forth. These statements typically include words such as *must*, *required*, *cannot*, or *valid*; they define the preconditions and invariants for the correct functioning of the operators.

In many cases, the processor detects violations of these restrictions and generates interrupts when violations occur. Some constraints are not enforced. Not all the checks applicable to the operator are mentioned in every operator description; many are described generally for a class of operator.

Restart Mode

When an interrupted code stream is resumed by repeating the interrupted operator, most operators can begin again to perform their normal function. Such operators are said to resume in initial mode. Sometimes, however, the resuming operator may not be able to start over from the beginning; for example, it might expect a different stack configuration or make different assumptions about some system state. For these operators, a restart mode is defined.

The fact that an operator resumes in restart mode provides one bit of information; when more is required, the interrupt mechanism leaves a word on the stack for the restarted operator to consume as an extra argument.

Restart mode occurs through accidental entry, as well as interrupts.

Restart mode is indicated by the rs bit in the stack lineage above an activation record; the bit is set as the procedure is entered by the interrupt mechanism or by accidental entry. The setting of the bit in these cases is determined by the operator and circumstances of the interruption. For explicit programmatic procedure entry, the bit is always reset. When an exit operator finds the bit set, the next operator executed is begun in restart mode.

In general, the information implied by the use of restart mode must be preserved until the operator is completed. Once an operator has been resumed in restart mode, any subsequent interruption and resumption of the same operation must use restart mode.

For details regarding the A16's use of restart mode, see "Restart Configuration" in Section 3.

Operator Applicability

Most of the operators defined in this manual are generally applicable; compilers may use them in programs written by and for end users. Some operators are restricted to special uses or are being phased out; several classes of operators are indicated in this manual:

1. **Unrestricted:** these operators are available for general use in user and system code files. Those operators that are new in MCP/AS (Extended) can be used only in code files to be run on such systems; most unrestricted operators may be used in "universal" code files or those that run in all supported systems.
2. **Obsolete:** these operators have been unrestricted, but they are no longer necessary and should not be used.
3. **Obsolescent:** these operators have been unrestricted and are needed in universal code files, but should not be used in code files compiled specifically for the A16 or newer systems.
4. **Partially restricted:** these operators have some application in user programs only in special situations. The partial restriction applies because the operators are dangerous if abused, but there is no cost-effective alternative to their use in some user-program constructs.

   In addition to the partially-restricted operators that are explicitly identified, there are restrictions on the application of other classes of operators that are not individually noted:

   a. Logical, bit-field, and similar operators must be applied with care to data types other than operands.

b. Operators containing address couples must be used to reference only data within the program domain and of the appropriate type. (Delta values must be within the range of the activation record; Lambda must be in {1 to LL} in user programs, although it is software convention, rather than the architecture, that places user code segment dictionaries at lex level 1 rather than 0.)

c. Any operator that can consume an arbitrary top-of-stack input must be used only against items in the expression stack and never to consume an original DD or stack-linkage word.

All partial restrictions are enforced by the compilers according to convention.

5. **Restricted:** these operators do not appear in user programs; they are dangerous and their use is restricted to low-level code in the operating system. (See Appendix E.)

Some operators, not otherwise restricted, are obsolete or obsolescent either because they have been superseded by more suitable operators or because they are deemed not to be cost-effective. Obsolete operators should not be used by compilers; there are better constructs on all supported systems. Obsolescent operators should no longer be used by compilers that are emitting code specifically for the A16 or newer systems.

There is a special class of obsolescence involving four operators that are being phased out of A Series systems. These operators (STAD, STAN, INXA, and NXVA) should not be emitted into code files intended for execution on the A16 or newer A Series systems. They will be executed by early A16 systems in order to migrate older code files to the A16. (See Appendix F for the definition of these operators.)

Obsolescence and restriction are noted in the individual operator specifications and are flagged in Appendix C.

Expression Stack Control

Most operators require items from the top of the expression stack and/or leave their results on top of the stack. Stack items required by operators are called arguments. They are normally consumed; that is, they are used and deleted from the stack. For consistency and brevity, this manual always defines arguments as deleted; any argument not consumed is defined as becoming an output.

Only the double precision operand (tag-2 pair) is treated in the expression stack as a single two-word item. All other data types are treated as single word items when encountered as arguments and are pushed or popped as single words.

The A16 may capture up to five expression-stack items in registers; the remainder are in memory in the current stack. The S register designates the topmost word in the memory stack; the processor automatically moves data between memory and the registers as necessary. The value of S is independent of any argument registers; that is, the register contents must be actually pushed into memory in computing S.

Items in registers are not addressable with address couples; a PUSH or enter operation is required to move the expression stack into the activation record.

The point at which Stack Overflow occurs is dependent upon the number of expression-stack registers and how they are used; the interrupt need not occur on the operator whose output moved virtual S beyond the stack. It is appropriate to consider this interrupt an asynchronous service exception.

Top-of-Stack Push Operations

Operators that produce top-of-stack results must *push* them, in order, onto the expression stack. Such a push operation only writes the memory stack if there is not enough room in the processor's TOS registers. If the topmost word of the stack is reached in a push operation, a Stack Overflow interrupt is generated. The interrupt does not occur between the two words of a double precision item or from the implicit Mark Stack through the Enter operation of an interrupt generation or accidental entry.

In processor state, a double precision operand is treated as a 96-bit operand with a single tag equal to two. When a tag-2 item is pushed onto the stack, the high order and low order words are written in that order, both with tags of two.

Top-of-Stack Pop Operations

Any operator that requires a stack argument must *pop* it from the expression stack. Such a pop operation only reads the memory stack if there are not enough items in the processor's TOS registers. If the word at the top of the memory stack has tag equal to two, the word below is also popped; if this word does not have tag equal to two, the A16 processor generates an Invalid Argument interrupt. The top and next words are taken as the second and first words of a double precision operand. (Note that because argument items may be either single or double words, multiple arguments must be accessed by popping them in order, from the topmost down.)

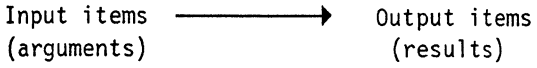
Operands, references, and other unprotected items that have been explicitly pushed into the addressing environment can be consumed as arguments, in which case, they automatically migrate back into the expression stack. Protected items such as original DDs, POWs, and stack linkage words do not migrate in this way, however PCWs do so that they can be deleted.

Stack Underflow is detected if a pop operation attempts to consume the stack linkage. Ideally, the lowest address that can be popped should be MAX(F+1,D[LL]+2) to protect both entered and incipient ARs. However, the A16 enforces only one of the checks, comparing S to F.

Stack-State Transformation

Stack state includes the top-of-stack items and the stack-addressing registers. Most operators take arguments from and place results on the stack without affecting any register except S.

Top-of-stack transformations are shown by diagrams that display inputs and outputs in order from top downward:



The effects of the transformation is that all the inputs are consumed and all the outputs are created. "Null ==>" and "==> Null" indicate that the operator has no inputs or outputs, respectively. There is an implicit variant. The item that was on the stack below the lowest input (if any) remains on the stack below the lowest output (if any), unaffected by the top-of-stack transformation of the operator. The input items are shown for the initial mode of the operator; some operators have additional stack input arguments in restart mode.

Initial stack items are designated Id: type(interpretation), where Id and interpretation are optional. Type may be a data type, as defined in Section 2 or a list of such types; *any* indicates no type restriction. An Id, if included, is a distinguishing name indicating how the item is to be used and/or establishing a reference for the final stack state. The generic names TOS and TOS2 represent the top-of-stack and second-from-top items. Interpretation is shown if the operator applies one of several possible interpretations of the type.

Final stack items are designated similarly. An Id or Id' indicates that the result is the named argument in identical or modified form, respectively; the modified argument has the same type unless another type is shown.

A simple description of inputs into outputs is inadequate for operators that modify stack linkages and registers. Schematic stack-state transformation diagrams illustrate these operations.

Descriptor Interpretation

Most of the data and all the code in A Series systems reside in memory segments described by VSDs. The address of a particular memory word is computed by adding an index to the base address of an actual segment. For data, the index and the reference to the base of the actual segment are combined into a single item, an IndexedDD.

The word referenced by an IndexedDD is located by adding the word index and the base address of the actual segment. The word index in an IndexedDD must be verified against the length field in the designated ASD. There is a special case: a descriptor designating the last+1 element of a page is an alternative to one designating the first element of the successor page. All operators that evaluate DDs must accept the alternative form as equivalent.

For the referenced word to be accessed, the segment must be present. The presence indication is in the ASD. Any attempt to access data in a nonpresent segment results in an Absent Segment exception.

Code is normally referenced via a PCW. (For edit-mode code executed as a result of an enter-table-edit operator, the reference is an IndexedDD.) The Make PCW operation replaces the address couple in the PCW with an ASD_number, so the PCW is a complete reference, analogous to the IndexedDD. Any attempt to enter or branch to a nonpresent code segment results in an Absent Segment exception.

Computational Operators

Operators in this group are loosely termed *computational operators* because they take arguments directly from the stack and leave some form of result on top of the stack.

Computational operators will not evaluate references; their arguments must be items on the stack initially. Required parametric values may be static code parameters or dynamic stack arguments.

Numeric Operand Interpretation

Operators in this group act on single or double precision operands interpreted as integers or floating-point numbers.

The leftmost bit of an operand plays no role in numerical computation. With certain exceptions defined for NTGR, numerical computation and type transfers leave bit 47 in an undefined state.

Representable Values

Consider the set V of all representable values. V is a subset of the rational numbers. An element of V can be represented by a 3-tuple (mi, mf, e), where:

mi is the integer part of the mantissa
mf is the fractional part of the mantissa
e is the exponent

The symbols +, -, *, / are used to denote the usual arithmetic operations on rationals and the symbol ** denotes exponentiation.

```
Let   p = maximum number of octal digits in mi;
      d = maximum number of octal digits in mf;
      q = maximum number of octal digits for the exponent;
      m = mi + mf * 8**(-d)
```

The set M of representable mantissas is

```
M = { m :  0 <= |mi| < 8**p,
           0 <= |mf| < 8**d and mf has the same sign as mi }
```

The set V of representable values is

```
V = { m  *  8**e : m in M, 0 <= |e| < 8**q }
```

```
For    p=13, d=0,  q=2, e=0:     V=I,    the set of representable single
                                         precision integers.

       p=13, d=0,  q=2:          V=S,    the set of representable single
                                         precision floating-point values.

       p=13, d=13, q=5, e=13:    V=Id,   the set of representable double
                                         precision integers.

       p=13, d=13, q=5:          V=D,    the set of representable double
                                         precision floating-point values.
```

A representation is defined to be normalized if |mi| >= 8**(p-1) or if mi = mf = e = 0. That is, a normalized representation is either zero in all components, or is non-zero in the high-order octade of the mantissa. The set Vn of normalized values is thus defined as a subset of V:

```
Vn = { v : v in V and ( 8**(p-1) <= |mi| < 8**p
                       or mi = mf = e = 0      ) }
```

Substitution of the appropriate values of p, d, and q into Vn defines the sets of normalized single and double precision, Sn and Dn. Some elements of V cannot be normalized, because |mi| < 8**(p-1) with the smallest representable exponent.

A representation is defined to be in normal form if it is normalized or has the smallest representable exponent: e = 1 - 8**q. The set V' of normal-form representations is defined as

```
V'  = { v : v in V  and  ( v in Vn  or  e = 1 - 8**q ) }
```

V' is a subset of the representations in V, however, every element of V has a unique normal representation in V'.

The representable values in V and its derived subsets are only a subset of the possible results of arithmetic operations upon elements of V. Let K be the set of all possible arithmetic results generated by the usual arithmetic operations on the elements of V. K is a subset of the rational numbers.

```
K =  { k : k=x+y, k=x-y, k=x*y, or k=x/y, y<>0, x,y in V }
```

V is a subset of K because for every v in V, v=v+0 is in K. Rounding and truncation functions are required to select a value in V to represent an element of K that is not in V. An element of K is not in V for two reasons:

1. It cannot be represented exactly with p+d digits of mantissa
2. Its absolute value is too large or too small to be represented with q digits of exponent.

In the former case, the number must be a value with fewer significant digits; in the latter case the number is out of range of the representation.

The notion of digit significance can be quantified in terms of the value of a "1" in the lowest-order digit of the mantissa. The unit value for any representation v = m * 8**e in V is u(v) = 8**(e-d). A rounding function must choose a value of v that is not more than u(v)/2 different from the value of k. A truncation function must choose a value of v such that |v| <= |k| < |v| + u(v). If there is no such value in V, the value of k is out of the range of V, so the rounding or truncation function indicates failure by choosing a value not in V. Two values are defined for this purpose, called TooBig and TooSmall. No arithmetic value is assigned to TooBig or TooSmall; it is convenient to define -TooBig = TooBig and -TooSmall = TooSmall.

Arithmetic Exceptions

Five arithmetic exceptions are defined. Precise conditions for their generation are specified later; the following is a summary of those conditions:

Exponent Overflow	exponent too large
Exponent Underflow	exponent too small with any nonzero mantissa
Precision Loss	exponent too small with normalized mantissa and denormalized result is inexact
Divide by Zero	divisor is zero
Integer Overflow	integer value too large for mantissa field

The first three exceptions always apply to the result of an arithmetic operation. Divide by Zero is an argument-value check in the division operators. Integer Overflow exception occurs when the output (or occasionally the input) value for a computation must be represented as an integer and cannot be represented as an integer.

The code stream can be resumed after an arithmetic-exception interrupt; the computation continues with a value specified for the exception or the operator.

Integer Overflow is defined as a resumable arithmetic exception that occurs when a computational result or the principle computational input for an operation must be represented as a single_integer, but is out of range. Another exception, Non-Integer Argument, is defined for the cases where an operator argument must be represented as a single_integer, but is out of range; this exception is generated for such arguments as index values, lengths, counts, and scale factors. The code stream cannot be resumed after an interrupt is generated to report a Non-Integer Argument.

There are three arithmetic interrupts for scalar operations and a fourth for vector operators. In each case, the input arguments and the operator are available to the interrupt routine, which is responsible for determining which exception occurred (if any) and for providing the defined resumption value. Certain *almost underflow* cases, which are not defined here as exceptions, can be reported by an arithmetic interrupt; the interrupt handling software is responsible for detecting these cases, producing the defined result, and resuming the code stream. Non-Integer Argument exceptions are reported, along with other infractions, as error conditions that are invalid or out of bounds arguments.

Rounding Function R

A rounding function R is defined below; R selects the element of V' that is closest to the desired value. If two elements of V' are equally close to the desired value, the element larger in absolute value is chosen.

The rounding function partitions the real line into intervals and maps each interval onto a representative element of that interval.

Example:

Consider

```
v = m * 8**e in V' (with (mi,mf,e) in normal form); let c = 1/2 * 8**(-d).
```

Then the intervals and their representative values are defined so that

v is the representative value for all x > 0 in the interval

```
                0 < lb(v) <= x < ub(v),

                ┌                                                  ┐
where lb(v) =   │ (m-c) * 8**e      m  not= 8**(p-1) or e = 1 - 8**q │
                │ (m-c/8) * 8**e  m = 8**(p-1)  and  e >  1 - 8**q │
                └                                                  ┘

and   ub(v) =   (m+c) * 8**e.
```

v is the representative value for all x < 0 in the interval

```
                -ub(-v) < x <= -lb(-v) < 0.
```

0 is the representative value for x = 0.

Consider the following examples, in which numbers are expressed as e,m in octal notation with d=0 and p=3.

```
a:    lb(v) =  0,122.4     v =  0,123     ub(v) =  0,123.4
b:    lb(v) = -1,777.4     v =  0,100     ub(v) =  0,100.4
c:    lb(v) = -1,776.4     v = -1,777     ub(v) = -1,777.4
```

Item "a" illustrates the *typical* case, in which (for e=0) the lower and upper bounds of the interval are m-1/2 and m+1/2, respectively. Item "b" illustrates the *anomalous* case where m = 8**(p-1), the bounds become m-1/16 and m+1/2 because normalization causes e to be smaller by one in lb(v) than in v. Item "c" illustrates that m = 8**p - 1 is a typical rather than an anomalous case; that is, an anomalous lb in one interval does not have a corresponding anomalous ub in the adjacent interval.

The rounding function for every k in K is defined as:

```
R:      K ⟶ V union { TooSmall, TooBig }

let     alpha =         c  * 8**(1 - 8**q)
        omega = (8**p - c) * 8**(8**q - 1)

        ┌                                                          ┐
        |  TooBig                    omega <= k                    |
        |  v             alpha <= lb(v) <= k < ub(v) <= omega      |
R(k) =  |  TooSmall                       0 < k < alpha             |
        |  0                                 k = 0                 |
        |  -R(-k)                            k < 0                 |
        └                                                          ┘
```

If the result of the rounding function is TooSmall, an Exponent Underflow exception is generated; if the code stream is resumed, the result of the operation is a zero value of the appropriate type (single or double precision).

If the result is TooBig, an Exponent Overflow exception is generated. If the code stream is resumed, the result of the operation is the largest representable magnitude of the appropriate type, with the appropriate sign.

R defines a class of rounding functions. Specific single and double precision rounding functions RS and RD are defined by selecting p, d, and q for R:

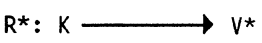
- For single precision, p=13, d=0, and q=2 define RS, which maps K onto (S union {TooSmall, TooBig}) with alpha = (1/2) * 8**(-63) and omega = (8**13 - (1/2)) * 8**63.
- For double precision, p=13, d=13, and q=5 define RD, which maps K onto (D union {TooSmall, TooBig}) with alpha = ((1/2) * 8**(-13)) * 8**(-32767) and omega = (8**13 - ((1/2) * 8**(-13))) * 8**32767.

In order to define when precision has been lost because the limited exponent does not permit the rounded result to be normalized, an extended rounding function, R*, is defined as follows:

Define the set V* of values with representable mantissas:

```
V* = { m * 8**e : m in M }
```

V* is a superset of V, defined by removing the exponent restriction. Now R* is defined to map



such that for every k in K, R*(k) is defined analogously to R(k), but with the m * 8**e values from V*. The definitions of v and c are the same for R* as for R, so lb(r) and ub(r) are also the same. However, because no limit q is applied, alpha is infinitesimal and omega is infinite.

If R(k) is in V, but the value of R(k) differs from R*(k), a Precision Loss exception is generated; if the code stream is resumed, the result of the operation is the representation of R(k).

Magnitude Round-to-Integer Function Ri

In order to integerize a value by rounding the absolute value, a special rounding function is necessary. This function, Ri, is analogous to R, but integer elements of V are chosen instead of elements of V'.

```
Ri:     K ------> V union { TooBig }  such that
        for every k in K, with integer v
                          and omega = 8**(d+p) - 1/2,

                 |  TooBig          omega <= k
Ri(k) =          |  v          0 < v - 1/2 <= k < v + 1/2 <= omega
                 |  0                 0 <= k < 1/2
                 |  -Ri(-k)                k < 0
```

If the result of the rounding function Ri is TooBig, an Integer Overflow exception is generated.

Ri defines a class of rounding functions. Specific single and double precision absolute value integer rounding functions are defined by selecting p, d, q, and e for Ri:

- For single precision, p=13, d=0, q=2, and e=0 define RI, which maps K onto (I union {TooBig}) with omega = 8**13 - 1/2.
- For double precision, p=13, d=13, q=5, and e=13 define RId, which maps K onto (Id union {TooBig}) with omega = 8**26 - 1/2.

The RId function is used whenever operators convert a value to a double precision integer by rounding. The RI function occurs only for the scale-right-round operators when the result is single precision.

Algebraic Round-to-Integer Function Rai

A second special rounding function, Rai, is defined to round an algebraic (signed) value to an integer. This function is the same as Ri, except that if two integers are equally close to the desired value, the algebraically larger integer is chosen.

```
Rai:    K ------> V union { TooBig }
        for every k in K, with integer v
                          and omega = 8**(d+p) - 1/2

                 | TooBig              omega <= k
Rai(k) =         | v        -omega <= v-1/2 <= k < v+1/2 <= omega
                 | TooBig                      k < -omega
```

If the result of the rounding function Rai is TooBig, a Non-Integer Argument exception is generated in all cases except the NTGR operator, for which Integer Overflow exception is explicitly specified.

The Ri and Rai functions are identical for v greater than or equal to zero, but form different sets of intervals for v is less than zero. For example, Ri(1.5) = Rai(1.5) = 2, but Ri(1.5) = -2, whereas Rai(-1.5) = -1. Note, also, that Rai(-omega) = -8**(p+d) + 1, but Ri(-omega) = TooBig. (Omega values are the same for Rai and Ra.)

Rai defines a class of rounding functions. Specific single and double precision algebraic integer rounding functions are defined by selecting p, d, q, and e for Rai:

- For single precision, p=13, d=0, q=2, and e=0 define RaI, which maps K onto (I union {TooBig}) with omega = 8**13 - 1/2.

The RaI function is used by NTGR and all operators that accept an operand and *integerize with rounding* (if necessary) to produce a single_integer for use as an index, iteration count, and so forth. No operators use the double precision algebraic round-to-integer function.

Truncation Function T

The truncation function T chooses the value in V' with the largest absolute value not greater than the absolute value of the desired value k. T partitions the real line into a different set of intervals from those of R.

Consider v = m * 8**e in V' (with (mi, mf, e) in normal form). Then the intervals and their representative values are defined such that:

v is the representative value for all x > 0 in the interval

```
0 < v <= x < succ(v),

    where      v  =  m             * 8**e
    and   succ(v) = (m + 8**(-d)) * 8**e.
```

v is the representative value for all x < 0 in the interval

```
-succ(-v) < x <= v < 0.
```

0 is the representative value for x = 0.

The truncation function T for every k in K is defined as:

```
T:      K ──────► V union { TooSmall, TooBig }

let     alpha = 8**(-d) * 8**(1 - 8**q)
        omega = 8**p    * 8**(8**q - 1)

                ┌                                                    ┐
                │ TooBig            omega <= k                       │
                │ v           alpha <= v <= k < succ(v) <= omega     │
T(k) =          │ TooSmall            0 < k < alpha                  │
                │ 0                       k = 0                      │
                │ -T(-v)                  k < 0                      │
                └                                                    ┘
```

If the result of the truncation function is TooSmall, an Exponent Underflow exception is generated; if the code stream is resumed, the result of the operation is a zero value of the appropriate type (single or double precision).

If the result is TooBig, an Exponent Overflow exception is generated; if the code stream is resumed, the result of the operation is the largest representable magnitude of the appropriate type, with the appropriate sign.

T defines a class of truncation functions. Specific single and double precision truncation functions are defined by selecting p, d, and q for T:

- For single precision, p=13, d=0, and q=2 define TS, which maps k onto (S union {TooSmall, TooBig}) with alpha = 8**(-63) and omega = 8**76.

The TS function is used (in conjunction with the N function) in the SNGT operator. No operators use the double precision truncation function.

Magnitude Truncate-to-Integer Function Ti

In order to integerize a value by truncation, a special integer truncation function is necessary. This function, Ti, is analogous to T, but integer elements of V are chosen instead of elements of V'.

```
Ti:     K ------> V union { TooBig } such that
        for every k in K with integer v
                  and omega = 8**(p+d):

                 | TooBig          omega <= k
Ti(k) =          | v             0 <= v <= k < v+1 <= omega
                 | -Ti(-k)                 k < 0
```

If the result of the truncation function Ti is TooBig, an Integer Overflow exception is generated.

Ti defines a class of truncation functions. Specific single and double precision absolute-value integer truncation functions are defined by selecting p, d, q, and e for Ti:

- For single precision, p=13, d=0, q=2, and e=0 define TI, which maps K onto (I union {TooBig}) with omega = 8**13.
- For double precision, p=13, d=13, q=5, and e=13 define TId, which maps K onto (Id union {TooBig}) with omega = 8**26.

The TId function is used whenever operators convert a value to a double precision integer by truncation. The TI function is used only by the NTIA operator.

Normalization Function N

A normalization function, N, is needed to put results in normalized form. Because Vn is a subset of V' and every element of V has a unique representation in V', every normalizable element of V has a unique normalized representation in Vn.

```
N:      V ------> Vn union { TooSmall } such that
for     alpha = 8**(p-1) * 8**(1 - 8**q) = 8**(p - 8**q),

        +-------------------------------------------+
        | v (normalized)            alpha <= |v|    |
N(v) =  | TooSmall                      0 < |v| < alpha
        | 0 (in normal representation)      v = 0   |
        +-------------------------------------------+
```

If the result is TooSmall, an Exponent Underflow exception is generated; if the code stream is resumed, the result of the operation is a zero value of the appropriate type (single or double precision).

N defines a class of normalization functions. Specific single and double precision normalization functions are defined by selecting p, d, and q for N:

- For single precision, p=13, d=0, and q=2 define NS, which maps V onto (Sn union {TooSmall}) with alpha = 8**(-51).
- For double precision, p=13, d=13, and q=5 define ND, which maps V onto (Dn union {TooSmall}) with alpha = 8**(-32755).

The composite functions R N and T N map the values into normalized representations and then perform rounding and truncation respectively. The following convert double to single precision:

```
RS ND: D --ND--> Dn union {TooSmall} --RS--> Sn union {TooSmall, TooBig}
TS ND: D --ND--> Dn union {TooSmall} --TS--> Sn union {TooSmall, TooBig}
```

The alpha value for ND is smaller than the alpha value for RS or TS, so that if ND returns TooSmall, the composite function returns TooSmall.

Approximation Functions

For arithmetic operations, a class of rounding functions R has been defined to map the set of arithmetic results, K, into the set

```
V union  {TooBig, TooSmall}
```

The definition of R does not depend on the fact that K is a set of rational numbers; the same function R can be applied to map the set K* onto the result set, where

```
K* = { k : k = f(x) or k = f(x,y), x,y in V
                                    and in the domain of f }
```

and f is a computable function of one or two variables.

As shown, R(f) could be used to define the correct value of the function. However, it may be impractical to evaluate mathematical functions with sufficient precision and accuracy to make such a definition useful. Instead, approximations are computed; rather than defining one correct result, the set of results that are sufficiently close to the correct one are defined.

Informally, the set of approximate values Aj(x) to the true value x includes those values that differ from x by no more than j times the value of the low-order bit, if the mantissa is a normal representation of x.

The set Aj(x) contains at most 2j+1 values near x. There is a *nearest number*, which is an element of Aj(x), defined by the Rounding function R(x). The other elements are the subset of v in V such that R(x + h*u(v')) = v, where v' is the representation of v in normal form, u is the unit function, and h is a nonzero integer in the range {-j to j}. It is possible for Aj(x) to contain only TooBig or TooSmall or to contain one of those values and up to 2j representable values.

The processor will choose an element of Aj(x) as its representation of x. Any element is acceptable. If TooBig or TooSmall is chosen, an Exponent Overflow or Exponent Underflow exception is generated, respectively.

Aj is a set-generating function based on the class of rounding functions R. The sets of single and double precision approximations ASj(x) and ADj(x) are defined by the specific rounding functions RS and RD, respectively.

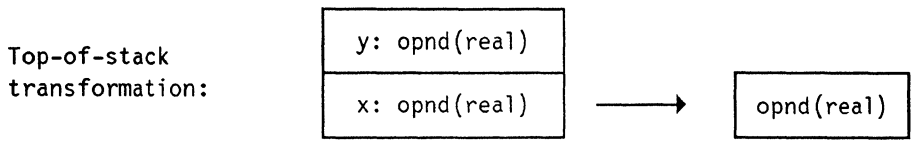
Note: *AS0(x) = {RS(x)} and AD0(x) = {RD(x)}*

If f(v) equals zero for some representable value v in V, then Aj(f(v)) equals zero. For example, square root must return zero for the argument 0.

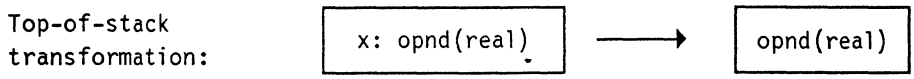
Numeric-Interpretation Operators

The operators in the following groups interpret operands numerically as a primary part of their function; numeric interpretation also occurs as some part of the function of operators in other groups.

For dyadic operators, the second from top of stack operand is denoted x and the top operand y. Unless a diagram is included in the operator definition, dyadic operators effect the following transition. Note that real interpretation includes integers as a subset.



For monadic operators, the top of stack operand is denoted x:



Arithmetic Operators

Arithmetic operators are dyadic, except for NORM, which is monadic.

Dyadic operators generate a single precision result if both operands are single precision and a double precision result if either or both operands are double precision. Where required, single precision is extended to double precision prior to the operation by appending a second word of all zeros; note that the numeric value of the operand is not changed.

Let AxB signify the Cartesian product of two sets A and B. The definition for the following arithmetic operators deals only with the domains SxS and DxD. Since S is a subset of D, the domains SxD and DxS are considered special cases of DxD.

The ranges of the operators are subsets of V, depending upon the values of p, d, and q and upon the rounding or truncation function invoked by the particular operator. If the result is TooSmall or TooBig, an interrupt appropriate to the operation is generated.

ADD <add>

ADD requires two operands on top of the stack. The numeric values of the two operands are algebraically added and rounded, and the result is left on top of the stack.

```
ADD:    S x S ⟶ S union { TooBig }
        such that x ADD y = RS(x+y) for x,y in S
or
        D x D ⟶ D union { TooBig }
        such that x ADD y = RD(x+y) for x,y in D but
        not x,y in S.
```

For x,y in S or D, 0 < |x ADD y| < alpha cannot occur, so Exponent Underflow is never generated by ADD. If both x and y are single_integers and the absolute value of the sum is less than 8**13, then the result is a single_integer.

SUBT <subtract>

SUBT requires two operands on top of the stack. The numeric value of the top item is algebraically subtracted from the numeric value of the second item and rounded and the result is left on top of the stack.

```
SUBT:   S x S ⟶ S union { TooBig }
        such that x SUBT y = RS(x-y) for x,y in S,
or
        D x D ⟶ D union { TooBig }
        such that x SUBT y = RD(x-y) for x,y in D but
        not x,y in S.
```

For x,y in S or D, 0 < |x SUBT y| < alpha cannot occur, so Exponent Underflow is never generated by SUBT. If both x and y are single_integers and the absolute value of the difference is less than 8**13, then the result is a single_integer.

MULT <multiply>

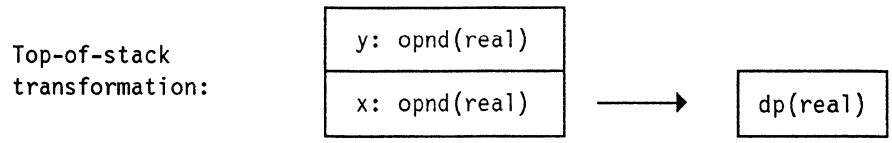
MULT requires two operands on top of the stack. The numeric values of the two operands are algebraically multiplied and rounded and the result is left on top of the stack.

```
MULT:   S x S ⟶ S union { TooSmall, TooBig }
        such that x MULT y = RS(x*y) for x,y in S,
or
        D x D ⟶ D union { TooSmall, TooBig }
        such that x MULT y = RD(x*y) for x,y in D
        but not x,y in S
```

If both x and y are single_integers and the absolute value of the product is less than 8**13, then the result is a single_integer.

MULX <extended multiply>

MULX requires two operands on top of the stack. Any single precision operand is extended to double precision before the numeric values are algebraically multiplied and rounded. The double precision result is left on top of the stack.



MULX over SxS is equivalent to a combination of XTND over S and MULT over DxD. MULX over DxD is equivalent to MULT over DxD.

DIVD <divide>

DIVD requires two operands on top of the stack. The numeric value of the second item is algebraically divided by the numeric value of the top item and rounded, and the result is left on top of the stack. If the divisor (top of stack operand) equals zero, a Divide by Zero exception is generated; if the code stream is resumed, the result of the operator is a zero of appropriate type (single or double precision).

```
DIVD:   S x S ⟶ S union { TooSmall, TooBig }
        such that x DIVD y = RS(x/y) for y < > 0
        and x,y in S,
```

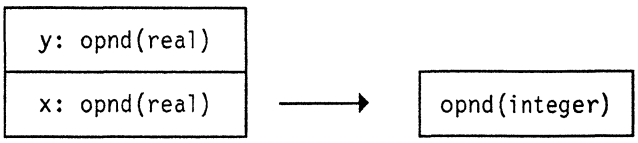
or

```
        D x D ⟶ D union { TooSmall, TooBig }
        such that x DIVD y = RD(x/y) for
        y < > 0 and x,y in D but not x,y in S.
```

IDIV <integer divide>

IDIV requires two operands on top of the stack. The numeric value of the second item is algebraically divided by the numeric value of the top item. The fractional part of the floating point quotient is discarded and the integer part is left on top of the stack in canonical integer representation.

Top-of-stack transformation:



If the divisor (top of stack operand) equals zero, a Divide by Zero exception is generated; if the code stream is resumed, the result of the operator is a zero of the appropriate type (single or double precision).

```
IDIV:   S x S ⟶ I union { TooBig }
        such that x IDIV y = TI(x/y) for y < > 0
        and x,y in S,
```

or

```
        D x D ⟶ Id union { TooBig }
        such that x IDIV y = TId(x/y)
        for  y  < > 0 and x,y in D but not x,y in S.
```

If an Integer Overflow exception occurs and the code stream is resumed, the result of the operator is an integer representation of appropriate type (single or double precision) and the value is the maximum integer of appropriate type and sign.

RDIV <remainder divide>

RDIV requires two operands on top of the stack. The numeric value of the second item is divided by the numeric value of the top item. The integer quotient with remainder is generated and only the remainder is left on top of the stack. The sign of the result is the same as the sign of the second item (the dividend).

If the divisor (top of stack operand) equals zero, a Divide by Zero exception is generated; if the code stream is resumed, the result of the operator is a zero of the appropriate type (single or double precision).

```
RDIV:   S x S ⟶ S union { TooBig }
        such that x RDIV y = x - (TI(x/y) * y)
        for y < > 0 and x,y in S,
or
        D x D ⟶ D union { TooBig }
        such that x RDIV y = x - (TId(x/y) * y)
        for y < > 0 and x,y in D but not x,y in S.
```

If both x and y are single_integers, then the result is a single_integer.

If an Integer Overflow exception occurs and the code stream is resumed, the result of the operator is of appropriate type (single or double precision) and the value is zero.

NORM <normalize>

NORM requires an operand on the top of the stack. If the operand is single precision, it is converted to normalized single precision representation. If the operand is double precision, it is converted to normalized double precision representation.

```
NORM:   S ⟶ S union { TooSmall }
        such that NORM x = NS(x) for x in S,
or
        D ⟶ D union { TooSmall }
        such that NORM x = ND(x) for x in D.
```

AMIN and AMAX <arithmetic minimum and arithmetic maximum>

AMIN and AMAX require two operands on top of the stack. The numeric values of the two operands are compared and the arithmetically lesser (or greater) of the two operand values is left as the result on top of the stack. If one of the input operands is single precision and the other input operand is double precision, the single precision operand is extended before the comparison and a double precision result is generated. If both of the inputs are single_integers, then so is the result.

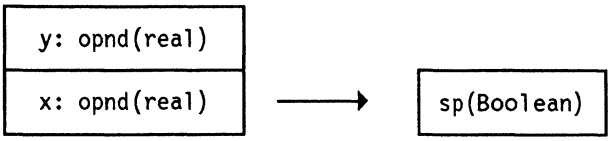
```
AMIN:   S x S ⟶ S   (x,y in S)  or
        D x D ⟶ D   (x,y in D but not x,y in S) such that

        x AMIN y = ⎡  x  if  x <= y ⎤
                   ⎣  y  if  x >= y ⎦

AMAX:   S x S ⟶ S   (x,y in S) or
        D x D ⟶ D   (x,y in D but not x,y in S) such that

        x AMAX y = ⎡  y  if  x <= y ⎤
                   ⎣  x  if  x >= y ⎦
```

Numeric Relational Operators

These six dyadic operators algebraically compare two real arguments to generate a Boolean result:

EQUL	<equal to>
NEQL	<not equal to>
LESS	<less than>
LSEQ	<less than or equal to>
GRTR	<greater than>
GREQ	<greater than or equal to>

Top-of-stack transformation:



The operators are defined for x,y in S union D. The result is True or False, expressing the truth of the proposition x : y, where : represents the relation expressed in the operator name.

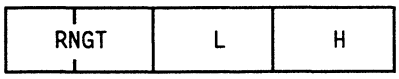
Range Test Operators

Range Test operators compute the result of a double arithmetic inequality, L <= X <= H. The value of X is a stack argument and is left on the stack along with the Boolean result.

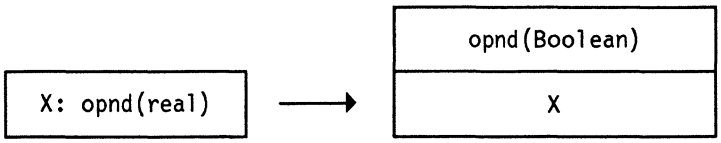
RNGT <range test>

The RNGT operator includes the values of L and H as two eight-bit parameters. It requires one operand (X) on top of the stack.

Operator format:

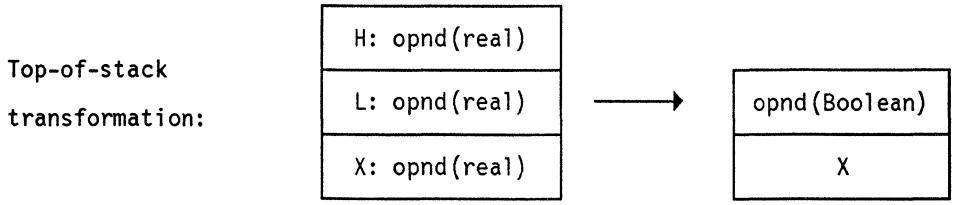


Top-of-stack transformation:



DRNT <dynamic range test>

The DRNT operator requires L and H as operands on the stack above X.



Numeric Type Transfer Operators

The following type transformations may be invoked on the top-of-stack item by operators defined in this group.

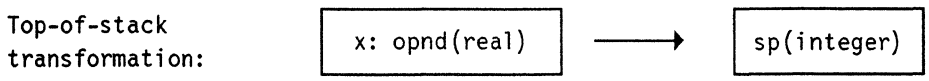
NTIA, NTGR:	Convert operand numeric value to single_integer.
NTGD, NTTD:	Convert operand numeric value to double_integer.
SNGT, SNGL:	Convert operand numeric value to single precision.
SNGT:	Convert WordDD to SingleDD

The following type transformations, among others, may be invoked on the top-of-stack item by operators defined in "Bit-Vector Type-Transfer Operators" in this section.

XTND:	Convert single precision operand to double precision or convert WordDD to DoubleDD.

NTGR <integerize rounded>

NTGR requires an operand on top of the stack. The operand is converted to single_integer representation and the result is left on top of the stack.




```
    NTGR:   S ⟶  I  union { TooBig }
or
            D ⟶  I  union { TooBig }
            such that NTGR x = RaI(x)  for x in S or D
```

If RaI(x) equals TooBig, an Integer Overflow exception occurs. If the code stream is resumed, the result of the operator is a single_integer and the value is 8**13-1 with appropriate sign.

If the argument is a nonzero single_integer, the result is the unmodified argument, therefore, bits [47:3] are preserved. For any other argument value, none of those three bits is set in the result if the corresponding bit is zero in the argument.

NTIA <integerize truncated>

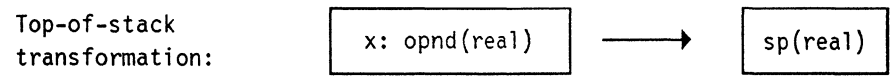
NTIA requires an operand on top of the stack. The operand is converted to single_integer representation and the result is left on top of the stack. The top-of-stack transformation is the same as for NTGR.

```
    NTIA:     S ⟶  I  union { TooBig }
or
              D ⟶  I  union { TooBig }
              such that NTIA x = TI(x)  for x in S or D
```

If an Integer Overflow exception occurs and the code stream is resumed, the result of the operator is a single_integer and the value is 8**13-1 with the appropriate sign.

SNGL <set to single precision rounded>

SNGL requires an operand on top of the stack. The operand is converted to normalized single precision representation and is left on top of the stack.




```
    SNGL:     S ⟶  Sn  union { TooSmall }
              such that SNGL x = NS(x) for x in S,
or
              D ⟶  Sn  union { TooSmall, TooBig }
              such that SNGL x = NS(RS(ND(x))) for x in D.
```

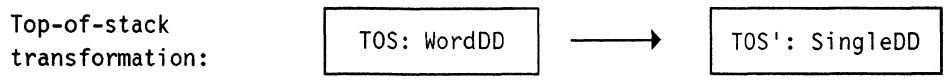
SNGT <set to single precision truncated>

SNGT requires an operand or copy WordDD on top of stack. An IndexedDD must have reindex enabled.

If the argument is an operand, it is converted to normalized single precision representation and left on top of the stack. The top-of-stack transformation is the same as for SNGL.

```
SNGT:    S ⟶ Sn  union { TooSmall }
         such that SNGT x = NS(x) for x in S,
or
         D ⟶ Sn  union { TooSmall, TooBig }
         such that SNGT x = NS(TS(ND(x))) for x in D.
```

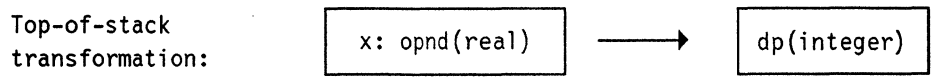
If the argument is SingleDD, it is left on the stack unchanged. If the argument is a DoubleDD, it is left on the stack as a SingleDD and the element_size is set to single precision.



SNGT applied to a WordDD is equivalent to SESW applied to the same argument; this use of SNGT is obsolescent.

NTGD <integerize double precision rounded>

NTGD requires an operand on top of the stack. The operand is converted with rounding to double_integer representation and the result is left on the top of the stack.




```
NTGD:    S ⟶ Id  union { TooBig }
         such that NTGD x = RId(XTND(x)) for x in S,
or
         D ⟶ Id  union { TooBig }
         such that NTGD x = RId(x) for x in D.
```

If the Integer Overflow exception occurs and code stream is resumed, the result of the operator is a double_integer and the value is 8**26-1 with appropriate sign.

NTTD <integerize double precision truncated>

NTTD requires an operand on top of stack. The operand is converted with truncation to double_integer representation and the result is left on top of the stack. The top-of-stack transformation is the same as for NTGD.

```
NTTD:   S ⟶ Id union { TooBig }
        such that NTTD x = TId(XTND(x)) for x in S,
or
        D ⟶ Id union { TooBig }
        such that NTTD x = TId(x) for x in D.
```

If the Integer Overflow exception occurs and code stream is resumed, the result of the operator is a double_integer; the value is 8**26-1 with appropriate sign.

Scale Left Operators

Scale left operators perform multiplication of an operand on top of the stack by ten raised to a power specified by a scale factor. The scale factor may be a dynamic argument or static parameter. These operators are obsolete; MULT by a power of ten is available.

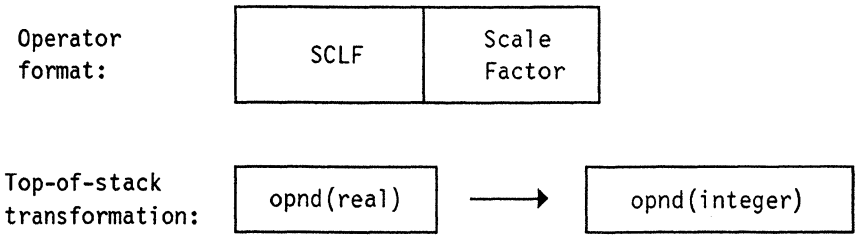
The item to be scaled must be an operand; if it is not an integer, it is integerized with the RId function. If the Integer Overflow exception occurs because RId equals TooBig and the code stream is resumed, the result of the scale-left operation is a double_integer; the value is 8**26-1 with appropriate sign.

If the scale factor is dynamic argument, it must be an operand; it is integerized with rounding (RaI function) if necessary. The scale-factor argument or parameter must be in the range {0 to 12}.

The result of the multiplication is left on top of the stack, represented as a single_integer or double_integer, depending on its magnitude. The result is single precision for the range {0 to 8**13-1} and double precision for {8**13 to 8**26-1}. If it is greater than or equal to 8**26, an indeterminate double precision integer is left on top of the stack and OFFF (overflow flip-flop) is set to 1; OFFF remains unchanged if the result is less than 8**26.

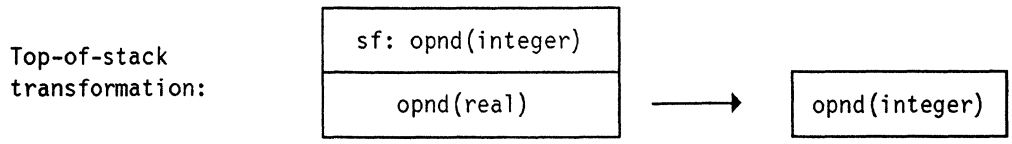
SCLF <scale left>

The top of stack operand is multiplied by ten raised to the power specified by the scale factor. The resultant single_integer or double_integer is left on top of the stack. The scale factor is a parameter:



DSLF <dynamic scale left>

The operand to be scaled is multiplied by ten raised to the power specified by the scale factor. The resultant single_integer or double_integer is left on top of the stack. Both arguments are required on top of the stack:



Scale Right Operators

Scale right operators perform division of an operand on top of the stack by ten raised to a power specified by a scale factor. The scale factor may be a dynamic argument or a static parameter. The results of the division are the quotient represented as a binary integer and/or the remainder represented as a decimal digit sequence. The operators that return a remainder (SCRF, DSRF, SCRS, DSRS) are obsolescent, superseded by BCD and DBCD.

The item to be scaled must be an operand. If necessary, it is integerized with the RId or TId function, depending upon the operator. If the Integer Overflow exception occurs because RId or TId equals TooBig and the code stream is resumed, the results of the scale-right operators have the following type, representation, and value.

SCRR, DSRR, SCRT, DSRT		double	integer	8**26-1 & sign
SCRF, DSRF		single	digit sequence	hex 999999999999
SCRS, DSRS	(TOS):	double	integer	8**26-1 & sign
	(2nd):	single	digit sequence	hex 999999999999

If the scale factor is a dynamic argument, it must be an operand; it is integerized (RaI function) if required. The scale-factor argument or parameter must be in the range {0 to 12}.

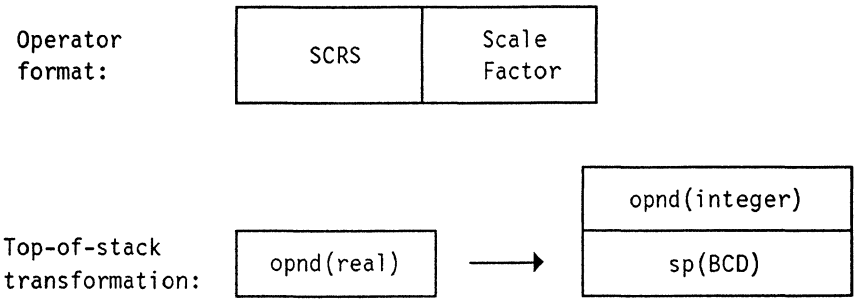
Scale right operators leave on top of the stack either the quotient of the division, the remainder, or both the quotient and remainder. The quotient is represented as a single_integer if its magnitude is in the range {0 to 8**13-1} and as a double_integer for the range {0 to 8**13 to 8**26-1}. (Note that the magnitude of the quotient cannot exceed 8**26-1.)

The remainder is a single precision operand interpreted as a left-justified digit (BCD) sequence. The number of digits in the remainder is equal to the scale factor and each digit is in the range {hex 0 to hex 9}. The values of the rightmost twelve - <scale factor> digits are undefined. The remainder is the unsigned result of dividing the absolute value of the integerized argument by a power of ten. In the following descriptions, x represents the argument to be scaled and n represents the scale factor.

SCRS <scale right save>

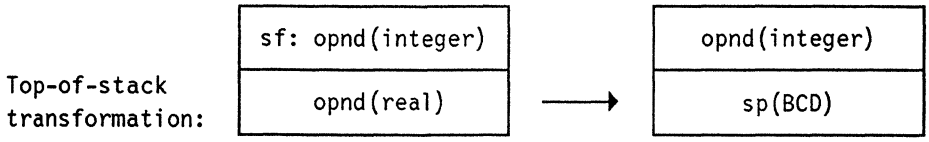
The argument to be scaled is integerized using the RId function. SCRS leaves the quotient on top of the stack and the remainder second from top of stack. The correct sign of the entire result is left in the sign bit of the quotient, even if the quotient itself is zero. EXTF (external sign flip-flop) is left in an undefined state.

The operand to be scaled is required on top of the stack and the scale factor is a parameter.



DSRS <dynamic scale right save>

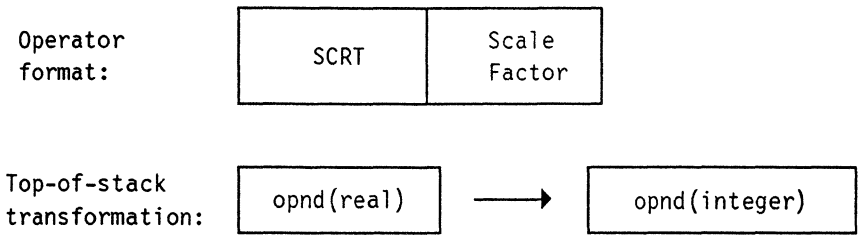
The operation is the same as SCRS, but the scale factor is required on top of the stack above the operand to be scaled:



SCRT <scale right truncate>

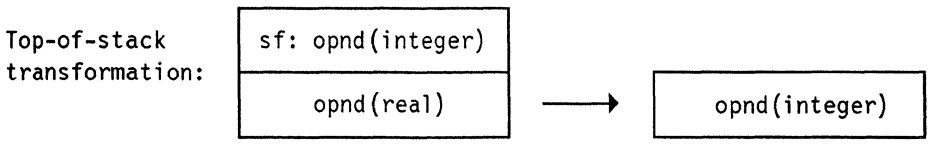
The argument to be scaled is integerized using the TId function. Only the quotient is left on top of the stack. (The operation effectively applies the generic Ti function to the quotient x/10**n.)

The operand to be scaled is required on the stack, and the scale factor is a parameter.



DSRT <dynamic scale right truncate>

The operation is the same as SCRT, however, the scale factor is required on top of the stack above the operand to be scaled:

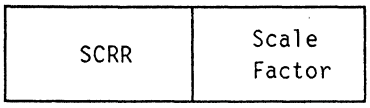


SCRR <scale right rounded>

The argument to be scaled is integerized, using the RId function if the scale factor is zero and the TId function otherwise. Only the quotient is left on top of the stack. If the most-significant digit of the remainder is greater than or equal to five, the magnitude of the quotient is increased by one. The operation effectively applies the generic Ri function to the quotient x/10**n. Note that if the rounded quotient is nonzero, it has the same sign as the original argument, even if the unrounded quotient is zero.

The operand to be scaled is required on top of the stack, and the scale factor is a parameter:

Operator format:



The top-of-stack transformation is the same as for SCRT.

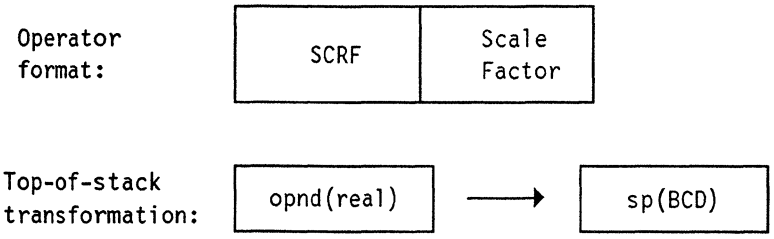
DSRR <dynamic scale right rounded>

The operation is the same as SCRR, however, the scale factor is required on top of the stack above the operand to be scaled; the top-of-stack transformation is the same as for DSRT.

SCRF <scale right final>

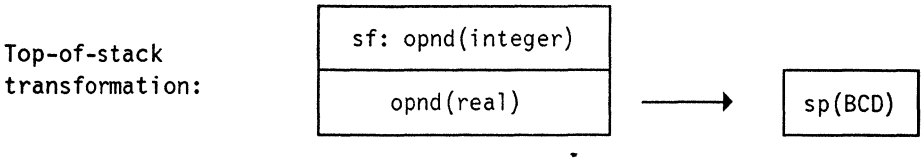
The argument to be scaled is integerized using the RId function. Only the remainder is left on top of the stack. EXTF (external sign flip-flop) is set to one if the mant_sign of the operand to be scaled is minus and to zero otherwise. OFFF (overflow flip-flop) is set to 1 if the quotient is any nonzero value; OFFF remains unchanged if the quotient is zero.

The operand to be scaled is required on top of the stack and the scale factor is a parameter:



DSRF <dynamic scale right final>

The operation is the same as SCRF, however, the scale factor is required on top of the stack above the operand to be scaled:



Binary To Decimal Conversion

The binary-to-decimal conversion operators are variations of the scale right final operators; the result is a decimal digit sequence representing the remainder of the division by a power of ten. There are two operators: a static form (BCD) with the number of digits specified as a code parameter and a dynamic form (DBCD) with the number of digits supplied as a stack argument.

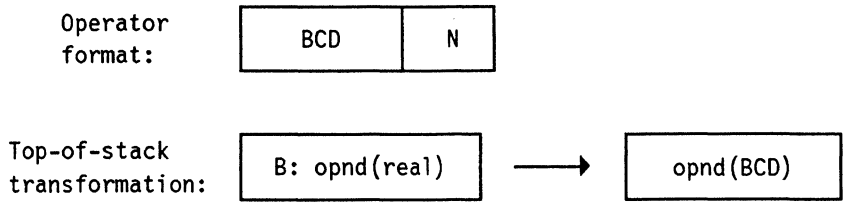
The number to be converted is a stack argument that must be an operand; if necessary, that argument is integerized with the RId function. This integerized argument is referred to in the next examples as B (the binary integer). If Integer Overflow exception occurs because RId equals TooBig and then the code stream is resumed, the result of the operation is single or double precision according to the value of N (as specified in the next paragraph) and the value is all nines.

|B| mod 10**N is converted to a sequence of N digits (each in the range (hex 0 to hex 9)), where N is provided as a code parameter or a stack argument. N must be in the range (0 to 24). The result is left-justified in an operand, which is single precision if N is 12 or less and double precision if N is in the range (13 to 24). The contents of the operand beyond the N-digit sequence are undefined.

```
EXTF is set to:  false (positive)   if    B  >= 0,
                 true  (negative)   if    B  <  0 and |B| mod 10**N > 0,
                 undefined state    if    B  <  0 and |B| mod 10**N = 0.
OFFF is set to:  true (overflow)    if   |B| >= 10**N,
                 unchanged          if   |B| <  10**N.
```

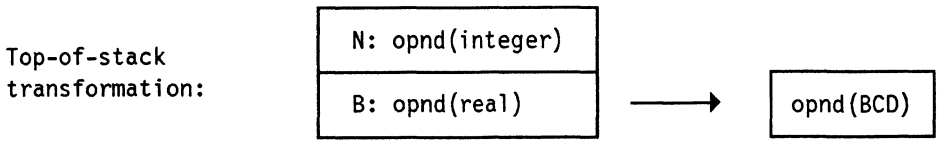
BCD <binary convert to decimal>

The B operand is the only stack argument; N is a code parameter:



DBCD <dynamic binary convert to decimal>

Two operand stack arguments are required:



The topmost argument is integerized with the RaI rounding function, if necessary, to produce N. The second argument provides B.

Mathematical Function Operator

RSQR <real square root>

RSQR produces single precision approximations if the input is single and produces double precision approximations if the input is double.

The argument must be an operand that is greater than or equal to zero.

```
RSQR: { s : s in S and s >= 0 } ⟶ S
      such that RSQR x in AS1(square root(x))
or
      { d : d in D and d >= 0 } ⟶ D
      such that RSQR x in AD1(square root(x))
```

The result of RSQR is greater than or equal to zero.

Bit Vector Interpretation

This group of operators provides various functions. Those operators that act on stack items either interpret the items as bit vectors or deal with the word as whole. In general, there are few restrictions on the type of stack items that are acted upon.

Logical Operators

Logical operators require one or two top-of-stack items. Either item may be of any type.

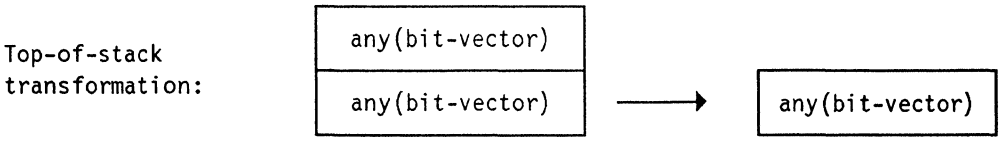
The items are interpreted as 48-bit vectors, unless one or both are double precision items. In that case, they are interpreted as 96-bit vectors and if only one of the two items is double precision, the other is extended with 48 zero bits (whether that item is an operand or not).

The logical operation is applied in parallel to each bit of the vectors and the result is left on top of the stack. For the monadic LNOT operator, the tag of the result is the same as the tag of the top of stack item. For the dyadic logical operators, the result is double precision if either argument is double precision; otherwise, the tag of the result is the tag of the second from top item.

The four logical operations are illustrated here in binary notation:

```
      NOT     01  =    10
0011  AND   0101  =  0001
0011   OR   0101  =  0111
0011  EQV   0101  =  1001
```

The dyadic logical operators effect the following transition:



LAND <logical and>

LAND requires two top-of-stack items. The logical AND of the two bit vectors is left on top of the stack.

LOR <logical or>

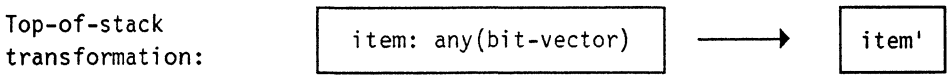
LOR requires two top-of-stack items. The logical OR of the two bit vectors is left on top of the stack.

LEQV <logical equivalence>

LEQV requires two top-of-stack items. The logical EQV (equivalence) of the two bit vectors is left on top of the stack.

LNOT <logical not>

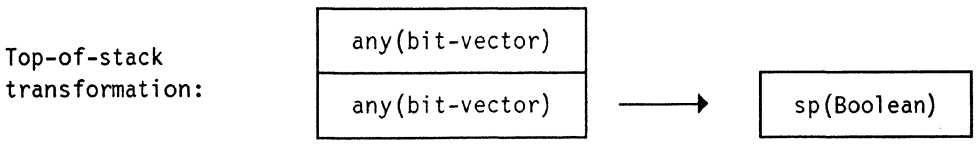
LNOT requires a single top-of-stack item. All bits of the vector are complemented and its tag remains unchanged.



Logical Relational Operator

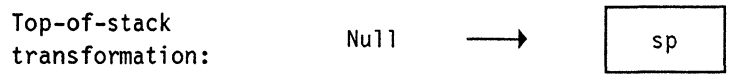
SAME <logical equality>

SAME requires two top-of-stack items. The items are interpreted as 52-bit vectors (including tag bits). If all corresponding bits of the two vectors have the same value, a True result is left on top of the stack; otherwise, a False result is left. If both items are double precision, the bit vector interpretation includes the second words. Note that if only one item is double precision, the result is necessarily false.



Literal Operators

Literal operators place a single precision constant on top of the stack:



ZERO <insert literal zero>

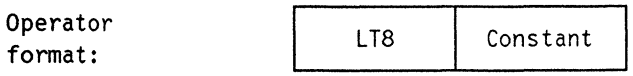
ZERO leaves on top of the stack a single precision word with all bits initialized to zero.

ONE <insert literal one>

ONE leaves on top of the stack a 1-bit integer equal to one.

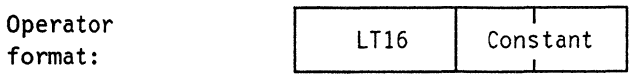
LT8 <insert 8 bit literal>

LT8 leaves on top of the stack an 8-bit integer that is a copy of its one-syllable parameter.



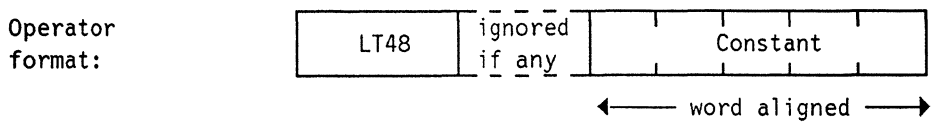
LT16 <insert 16 bit literal>

LT16 leaves on top of the stack a 16-bit integer that is a copy of its two-syllable parameter.



LT48 <insert 48 bit literal>

LT48 leaves on top of the stack a single precision operand that is a copy of its six-syllable parameter. The parameter is taken from the first code word following the LT48 op code. Padding syllables, if any, from the op code to the end of the word containing the op code are ignored.



Bit-Vector Type-Transfer Operators

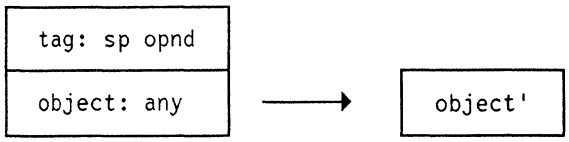
Operators in this group perform the following operations on the top-of-stack items.

STAG	Set the tag to an arbitrary value from the top of stack
XTND	Append a low-order word of zeros, if necessary, to form a double precision operand, or set the element_size of a wordDD to double precision
JOIN	Join two operands to form one double precision operand
SPLT	Split an operand into two single precision operands

STAG <set tag>

STAG requires a tag value and an object item on top of the stack and leaves as its result an item whose tag is the tag value and whose 48 bits are copied from the object item.

Top-of-stack transformation:



The topmost argument must be a single precision operand. The tag value is extracted from field [3:4]; the remainder of the operand is ignored.

If the tag value is 2 and the object item does not have tag 2, then the least-significant word is set to zero.

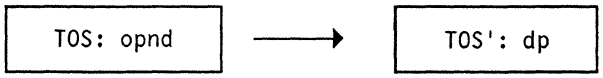
The STAG operator is partially restricted; in user code, the only valid tag values are 4, 5, and 6.

XTND <set to double precision>

XTND requires an operand or a copy WordDD on top of the stack. An IndexedDD must have reindex enabled.

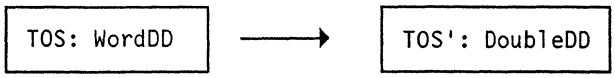
If the argument is a double precision operand, it is left on the stack unchanged. If it is single precision operand, it is converted to double precision representation by appending a second word whose fields are initialized to zero; the double precision result is left on the stack. Note that its numeric value is not changed.

Top-of-stack transformation:



If the argument is a DoubleDD, it is left on the stack unchanged. If the argument is a SingleDD, it is left on the stack as a DoubleDD and its element_size is set to double precision.

Top-of-stack transformation:

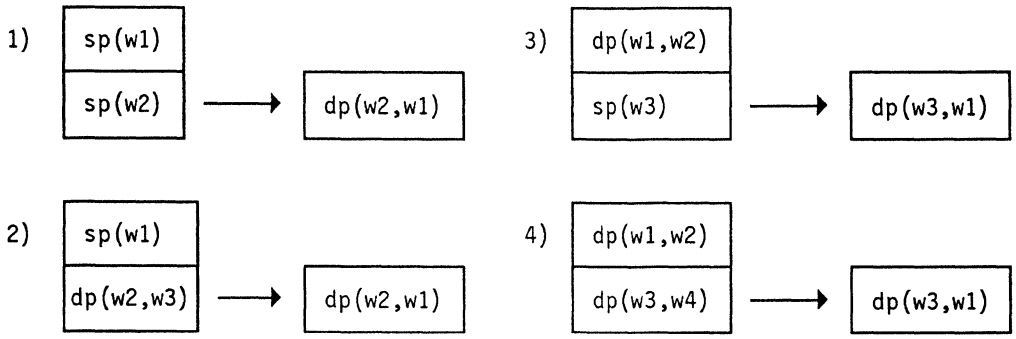


XTND applied to a WordDD is equivalent to SEDW applied to the same argument; this use of XTND is obsolescent.

JOIN <join two operands into double>

JOIN requires two operands on top of the stack. A double precision item is constructed from the two operands and the result is left on top of the stack.

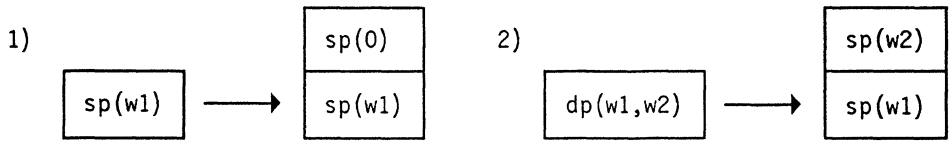
The first and second words of the double precision result are taken from the first words of the second and top operands, respectively. The following possibilities arise from combinations of single and double precision operands:



SPLT <split operand into two singles>

SPLT requires an operand on top of the stack. Two single precision items are constructed from the operand and left on top of the stack.

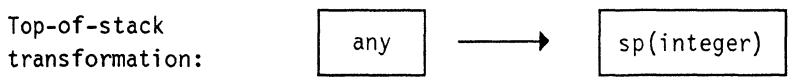
If the operand is single precision, it is left on the stack and a single precision zero is pushed on the stack above it. If the operand is double precision, its two words are converted to two single precision items. The first word is pushed on the stack first and the second word is left on the top of the stack.



Evaluate Word Structure Operators

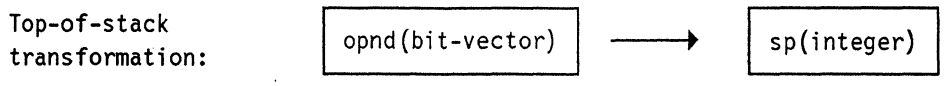
RTAG <read tag>

RTAG requires one item on top of the stack; its result is a single_integer whose value is the tag of the item.



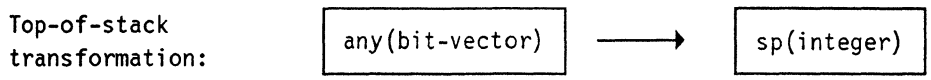
CBON <count binary ones>

CBON requires an operand on top of the stack. The number of bits in the operand that have the value one are counted. If the operand is double precision, all 96 bits are examined. CBON leaves on top of the stack a 7-bit integer whose value is the ones count.



LOG2 <leading one test>

LOG2 requires one item on top of the stack and leaves in its place a 6-bit integer that indicates the leading bit in the item having the value one. If all bits in the item are zero, LOG2 leaves an integer zero; otherwise, the integer value is the bit number plus one of the highest-order one bit. Only the first word of a double precision item is examined.



Word Manipulation Operators

Word manipulation operators provide the capability to alter any *partial field* of a word in the stack called the destination; in some cases this is based on a field of another word in the stack called the *source*. The source can be an argument or an implicit bit value; the destination can be an argument or an implicit word of zeros. The altered destination item is left on the top of the stack. The following operations are provided:

BSET, DBST	Set a single destination bit
BRST, DBRS	Reset a single destination bit
ISOL, DISO	Create a destination whose low-order field is set from a field of the source
INSR, DINS	Set a field of the destination from the low-order field of the source

FLTR, DFTR	Set a field of the destination from a field of the source
CHSN	Complement the *sign bit* (bit 46) of the destination

Source items may be of any type. Destination type is restricted only for CHSN, which requires an operand.

If the source is a double precision item, the field is taken from its first word and the second word is discarded. If the destination is a double precision item, the bit or field altered is in its first word and the second word is retained unchanged in the double precision result. The following terms are used for bit/field specifications:

Db	The destination bit to be set or reset or be the high-order bit of the destination field
Sb	The high-order bit of the source field
Len	The length of both the source and destination fields

There are static and dynamic operators corresponding to several of the operations. The static operators take Db, Sb, and Len specifications, as required, from the code parameters; the dynamic operators take them from stack arguments.

Dynamic Db, Sb, and Len arguments must be operands; they are integerized with rounding (RaI function) if necessary. All Db and Sb values must be in the range {0 to 47} and Len values must be in {0 to 48}.

The effect of word manipulation operators are shown as an assignment to a field of the destination word. The remainder of the destination word is not changed. Note that Len equals zero is a valid specification of a null field; in this case, the destination is not to be altered.

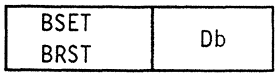
BSET <bit set>
BRST <bit reset>

BSET sets a single destination bit: destination.[Db:1] := 1.

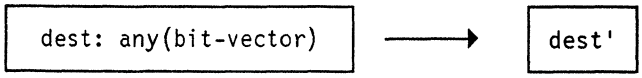
BRST resets a single destination bit: destination.[Db:1] := 0.

The destination is the only required top of stack item and Db is specified by a parameter.

Operator format:



Top-of-stack transformation:



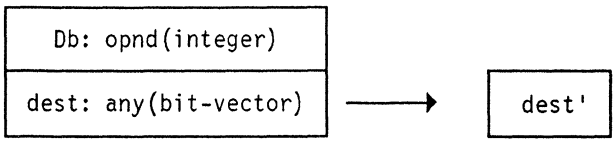
DBST <dynamic bit set>
DBRS <dynamic bit reset>

DBST sets a single destination bit: destination.[Db:1] := 1.

DBRS resets a single destination bit: destination.[Db:1] := 0.

The required initial stack state includes Db.

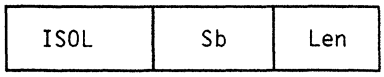
Top-of-stack transformation:



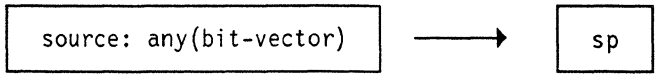
ISOL <field isolate>

ISOL creates a single precision destination word initialized to zero and then sets its low-order field from a field of the source: destination := 0; destination.[Len-1:Len] := source.[Sb:Len]. The source is an argument; Sb and Len are specified by parameters:

Operator format:



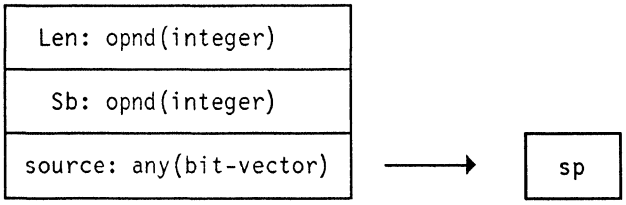
Top-of-stack transformation:



DISO <dynamic field isolate>

DISO creates a single precision destination word initialized to zero and then sets its low-order field from a field of the source: destination := 0; destination.[Len-1:Len] := source.[Sb:Len]. Sb and Len are arguments above the source.

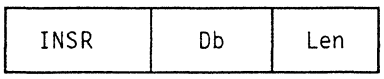
Top-of-stack transformation:



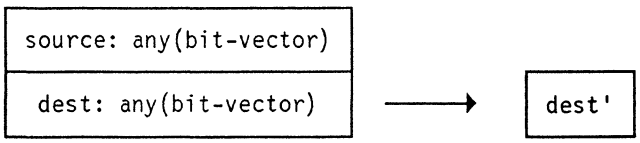
INSR <field insert>

INSR sets a field of the destination from the low-order field of the source: destination.[Db:Len] := source.[Len-1:Len]. The destination and source are arguments; Db and Len are specified by parameters:

Operator format:



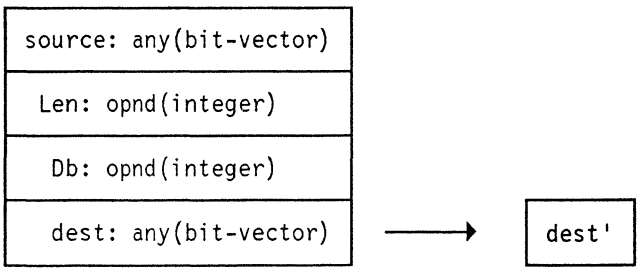
Top-of-stack transformation:



DINS <dynamic field insert>

DINS sets a field of the destination from the low-order field of the source: destination.[Db:Len] := source.[Len-1:Len]. Db and Len are arguments between the destination and source; note that the source is topmost:

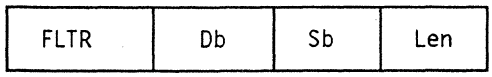
Top-of-stack transformation:



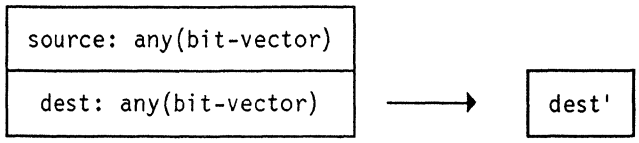
FLTR <field transfer>

FLTR sets a field of the destination from a field of the source: destination.[Db:Len] := source.[Sb:Len]. The destination and source are arguments; Db, Sb, and Len are specified by parameters:

Operator format:



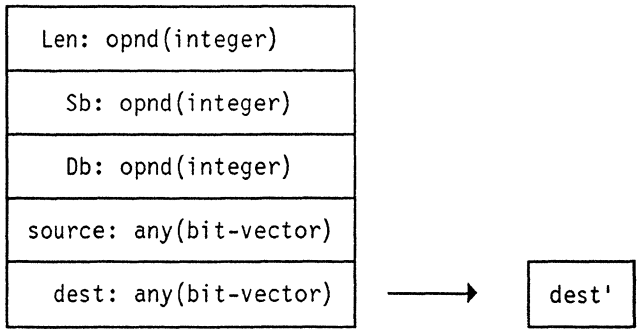
Top-of-stack transformation:



DFTR <dynamic field transfer>

DFTR sets a field of the destination from a field of the source: destination.[Db:Len] := source.[Sb:Len]. Db, Sb, and Len are arguments above the destination and source:

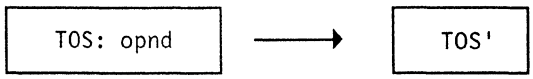
Top-of-stack transformation:



CHSN <change sign>

The CHSN operator requires an operand on top of the stack. CHSN complements a single destination bit: destination.[46:1] := NOT destination.[46:1].

Top-of-stack transformation:



Linear Index-Function Operator

OCRX <occurs index>

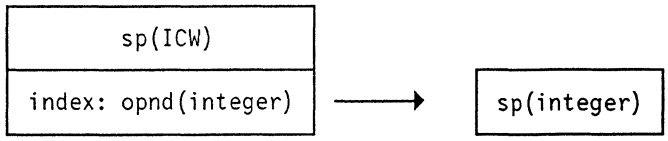
OCRX computes a linear integer function of an integer index, with bounds checking. The function is defined as:

```
RelativeIndex (offset, width, index) = offset + (index-1)*width;
```

where index must be in the range {1 to limit}.

OCRX leaves on top of the stack the result of the RelativeIndex function applied to values derived from two arguments:

Top-of-stack transformation:



The Index Control word (ICW) must be a single precision operand. It contains three fields:

```
ICW_width     [47:16]   =   the width coefficient
ICW_limit     [31:16]   =   the upper bound for the index
ICW_offset    [15:16]   =   the offset coefficient
```

The index argument must be an operand; it is integerized with rounding (RaI function) if necessary.

If the index is not in the range {1 to ICW_limit}, an interrupt is generated. Otherwise, ORCX leaves on top of the stack a single_integer whose value is RelativeIndex (ICW_offset, ICW_width, index).

Reference Generation and Evaluation Operators

The common feature of this operator group is the generation and evaluation of references and chains of reference. The group consists of reference generation operators (generators) and operators that evaluate references in order to read a target item onto the stack (read evaluators) or store an item from the stack into a target location (store evaluators).

Basic evaluation of a reference consists of calculating the absolute address to which it refers. Read evaluation consists of fetching the contents of the referenced location; store evaluation consists of writing the contents of that location.

Double Precision

References to double precision operands refer to the first word. The need for a second word may be indicated by the tag of the first word or by the reference (IndexedDoubleDD). The second word is the *virtual successor* of the first; it may be in the next higher memory location or the two words may be the last word of one page and the first word of the next page in the same virtual segment. Such separation can occur only when access is via an IndexedDoubleDD with an odd index value.

Stack References

A stack is designated by its stack_number. Each stack is an actual segment whose ASD_number is its stack_number.

Environment Link Evaluation

The basic evaluation of an environment link consists of computing the absolute address of the base of the referenced AR by adding the displacement to the base address of the referenced stack or non-stack segment.

Environment links occur in SIRWs and MSCWs. Those in SIRWs are evaluated as the SIRW is evaluated. Those in MSCWs are evaluated when the environmental chain is traversed to update display registers.

Evaluation of References

Read and store evaluators share the general capability to process a reference or chain of references in order to locate some target item. Initial references may be address couple parameters, SIRWs, IndexedWordDDs, sometimes ASRWs, and (for LODT) integers. Reference chains may contain SIRWs, IndexedWordDDs, and PCWs.

Definition of valid target items and allowable reference chains depends on the function of the particular operator; evaluation of each element of a reference chain and of IRW chains is common to the operator group. The following paragraphs define evaluation of each reference form, IRW chain evaluation, and the notation used for each operator to specify allowable reference chains and valid target items.

Address Couple Evaluation

Address couples occur in operator parameters in either fixed or variable fence forms. A name-call operator uses its address-couple parameter to construct an SIRW on the stack. Other operators use their address-couple parameter as their own initial reference.

Basic evaluation of an address couple (lambda,delta) consists of calculating the address of the referenced word, by adding delta to the base address of the activation record whose lex level is specified by lambda. For lambda equal to LL, the base address is in D[LL]; for lambda less than LL, the activation record is located LL minus lambda steps down the environmental chain (its base address is captured in D[lambda]).

In a valid address couple, lambda is less than or equal to LL and delta is less than the width of the referenced AR. For the topmost AR, the upper boundary includes only items that have been explicitly pushed, not items in the expression stack. The restrictions of this paragraph are incumbent upon the compilers for all A Series. The A16 enforces the rule that lambda is less than or equal to LL.

SIRW Evaluation

Basic evaluation of an SIRW consists of calculating the address of the referenced word; SIRW.offset is added to the address derived from the Environment Link. (The ENTR operator uses the Environment Link address, as well as the sum.)

No validity check is required on the magnitude of the offset value or the environment-link displacement value during SIRW evaluation. SIRWs are created from the address couple whose lambda is verified at that time. (If software constructs or modifies an SIRW, it is trusted to do so correctly.)

IndexedWordDD Evaluation

Basic evaluation of a data descriptor as a reference is possible only for an IndexedWordDD referring to a present segment. The evaluation consists of calculating the address of the referenced word.

Access checking is performed upon evaluation of IndexedWordDDs. A store is aborted if write is disabled via the DD.

In cases where the target of an IndexedWordDD is an operand, the element type of the operand is determined by the element_size field of the IndexedWordDD. The element_size value of single or double precision overrides the tag of the target operand. All operators that evaluate IndexedWordDDs obey this convention except the Load Transparent, overwrite, and transfer words operators.

An IndexedSingleDD may be used to reference words of any type appropriate to the referencing operator. An IndexedDoubleDD may be used only to reference an operand; the exception is LODT.

PCW Evaluation

In the context of reference chain evaluation, evaluation of a PCW consists of an accidental procedure entry. The PCW is assumed to point to a function with no parameters, whose returned value is either the target of the chain or another valid reference. The net result of the accidental entry is that the place of the PCW in the reference chain is taken by the item returned by the function.

The accidental entry is accomplished by the aACCE action. The function must terminate with a RETN (return) operator, leaving an item on top of the stack.

Reference Chains

Operators that evaluate reference chains start from an initial reference and apply successive reference read evaluation, according to a set of chaining rules, until a target item is produced. For each of these operators, the set of initial references and targets is specified using the following notation:

```
<Initial Reference> ::= {set of reference items}
<target>            ::= {set of target items}
```

Chaining rules are specified by indicating the valid evaluation results for each reference form that may be part of the chain. The form of such specification is:

```
reference ⟶ evaluation results
```

where the arrow indicates read evaluation of the reference as defined in the preceding paragraphs. Evaluation results can include a reference, which is subsequently evaluated, or a target.

Chaining rule notation is illustrated by the next example. IndexedWordDD is used for the union of IndexedSingleDD or IndexedDoubleDD, which are expanded separately.

```
<Initial Reference>   ::=  {SIRW, IndexedWordDD}
<target>              ::=  {operand}

SIRW           ⟶  SIRW
                  IndexedWordDD
                  PCW
                  <target>

IndexedWordDD  ⟶  <target>

PCW            ⟶  SIRW
                  IndexedWordDD
```

The term IRW denotes a reference in {address couple, SIRW}. The term *IRW chain* denotes a reference chain whose initial reference is an address-couple parameter or SIRW and that otherwise contains only SIRWs; an *SIRW chain* contains only SIRWs, including the initial reference.

Evaluation of the chain continues until a target is encountered or an interrupt occurs.

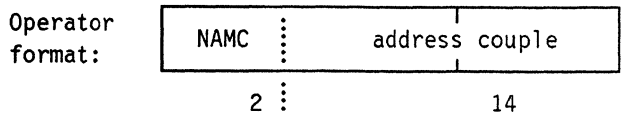
- Touch Reference (or Touch Load or Missing PCW) and Binding Request interrupts are generated if the target of any reference is an untouched DD or a DD with element_size = 7, respectively. Upon resumption after the interrupt, evaluation continues; the incorrect object has presumably been replaced by another reference or a target.
- Absent Segment interrupt is generated if a reference designates a nonpresent segment; evaluation continues when the code stream is resumed with the segment present.
- If an item is encountered that is not a valid result and not one of the service requests previously noted, an error interrupt is generated.

A reference chain must not loop, because the operator cannot complete. (The processor generates a Loop Timer alarm interrupt whenever an operator does not complete in a reasonable time.)

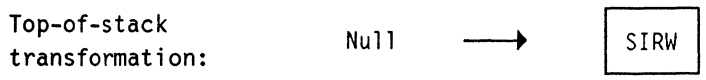
Reference Generation Operators

NAMC <name call>

On the A16, the NAMC operator transforms an address couple in the code stream into an SIRW. NAMC is a two-syllable operator with a special structure, a 2-bit op code and a 14-bit variable-fence address couple:

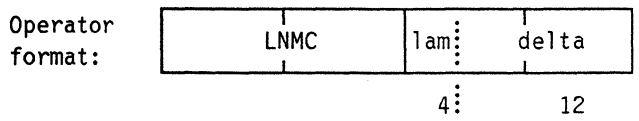


The SIRW is constructed to point to the word in the stack referenced by the address couple (lambda,delta). The offset field is set from delta. The env_link field is set to designate the normal or pseudo activation record at lex level lambda in the current addressing environment. Since the display registers are maintained as (ASD_number, displacement) couples, the Environment Link value is in D[lambda].



LNMC <long name call>

LNMC is equivalent to NAMC, except that its parameter is a fixed-fence rather than a variable-fence address couple. LNMC is a four-syllable operator whose appearance in the code stream is:



STFF <stuff>

The A16 performs no action.

aAPIX <apply index to descriptor>

This common action is defined as a procedure with two formal parameters, a descriptor DD, and an index value vI (a number of elements of the same size defined by the DD element_size). The action is to index DD by vI. The clients of aAPIX and their invocations are as follows:

- The indexing operators pass DDs, both indexed and unindexed, paged and unpaged.
- All array operators reindex indexed DDs via aAPIX.

The aAPIX action constructs an IndexedDD whose virtual index is

```
vX = vD + vI*df
```

where

vX is the new virtual index
vD is the virtual index of the DD (0 for unindexed DD)
vI is the argument to aAPIX
df is 2 if DD is double precision and 1 otherwise

For application to a DD, the vX must be represented in up to three components: page index (pX), word index (wX), and character index (cX); these components are defined as follows:

```
unpaged (pX not relevant):

single or double:            wX = vX
hex:     cX = vX MOD 12;     wX = vX DIV 12
EBCDIC:  cX = vX MOD 6;      wX = vX DIV 6

paged:                                                  (p represents 2**13)

single or double:            wX = vX        MOD p;  pX = vX        DIV p
hex:     cX = vX MOD 12;     wX = vX DIV 12 MOD p;  pX = vX DIV 12 DIV p
EBCDIC:  cX = vX MOD 6;      wX = vX DIV 6  MOD p;  pX = vX DIV 6  DIV p
```

When pX = n > 0, wX = 0, and (if applicable) cX = 0, the same element can be designated with pX = n-1 and wX = 2**13. A descriptor can designate the last+1 element of a page instead of the first element of the successor page. This alternative representation can be generated for all indexing; it is the only valid representation of the last+1 element in a paged virtual segment whose last page is 2**13 words long. The A16 chooses the alternate representation for pointer op, INDX, and SE8C & SE4C clients whenever the index change (vI) is algebraically greater than zero or if the DD is already in the alternate representation and the index change (vI) is zero.

The quotient pX is used to index the page table. For indexing a paged unindexed DD, the DD itself provides the reference to the page table. For reindexing an indexed DD that designates a page, a reference to the page table is extracted from the ASD designated by the DD. Each page ASD contains a reference to the original SingleDD for that page; this reference is an indexed copy of the page-table DD.

The new virtual index must be representable in an IndexedDD if the aAPIX client uses or returns the result; it is not necessary if the indexing operation is performed in a non-update Data Array operator in anticipation of an iteration that never occurs. The representable values have:

```
vX >= 0
pX <  Pages (for paged DDs)
wX <  field limit

where  Pages        = page-table length
  and  field limit  = 2**20 for WordDDs, 2**16 for CharDDs.
```

Pages is defined as ASD.AS_length in the ASD for the page table.

Valid representations have wX less than or equal length in words of the designated actual segment. Equality is expressly permitted; indexing can generate an IndexedDD to the first element beyond the array.

For the A16, if the aAPIX result is representable but not valid (it designates an element beyond last+1), an INDX or skip operator aborts. However, software should not depend upon the INDX or skip operators completing or aborting for such index values.

If the data segment is unpaged, an IndexedDD is constructed by making a proper copy of the DD, marking it indexed, and inserting the index. For a WordDD, an IndexedWordDD is constructed by setting the index field to wX with the field-width limit at 2**20-1. For a CharDD, an IndexedCharDD is constructed by setting the word_index field to wX and the char_index field to cX with the field-width limit on wX at 2**16-1.

If the data segment is paged, an IndexedDD is constructed by making a proper copy of the page descriptor, which is located by indexing the page table by pX. The element_size field is copied from the indicated DD. An IndexedWordDD is constructed with wX in the index field or an IndexedCharDD is constructed by setting word_index to wX and char_index to cX; field overflow cannot occur.

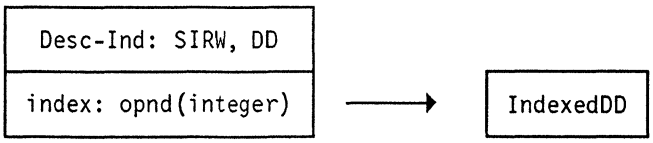
INDX <index>

INDX applies an integer index to a DD and leaves on top of the stack an IndexedDD pointing to the specified element. If the DD is a WordDD, the result is an IndexedWordDD and if it is a CharDD, the result is an IndexedCharDD. A copy DD may be on the stack initially; an original or copy DD may be addressed by an SIRW chain.

```
<Descriptor Indication> ::= {unindexed copy WordDD,
                             unindexed copy CharDD,
                             IndexedWordDD,
                             IndexedCharDD,
                             initial reference}

<Initial Reference>     ::=  SIRW

<target>                ::= {unindexed WordDD, unindexed CharDD
                             IndexedWordDD,
                             IndexedCharDD }

SIRW                    ---> SIRW
                             <target>
```

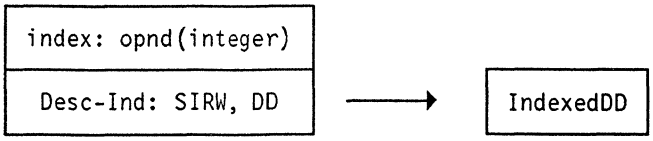
INDX requires the Descriptor Indication and an operand index value on top of the stack in either order.

Top-of-stack transformation:



or

Top-of-stack transformation:



The index value must be an operand. The Descriptor Indication must be an SIRW or a copy WordDD or Char DD. In describing the indexing operation, the unqualified term DD is applied to the indicated descriptor, the one to which the index will be applied. If the Descriptor Indication is a data descriptor, it is the DD; if the Descriptor Indication is an SIRW, the DD is the target of the SIRW chain.

The index value is integerized with rounding (RaI function) if necessary; its magnitude must not exceed MaxInx. The resulting value is called I within this description.

The DD may be unindexed or indexed; an indexed DD must have reindex enabled. The new virtual index is the sum of the current virtual index of DD (which is 0 if DD is unindexed) plus I; the sum must be in the range {0 to N}, where N is the number of elements in the virtual segment referenced by DD. (This constraint can be taken as a summary of the detailed constraints defined in aAPIX.)

The index value, I, is applied to the descriptor DD using the common action aAPIX; INDX aborts if the indexing operation does not produce a valid descriptor.

There are two access rights in an IndexedDD: write enable and reindex enable. The indexing operation determines the access rights as follows:

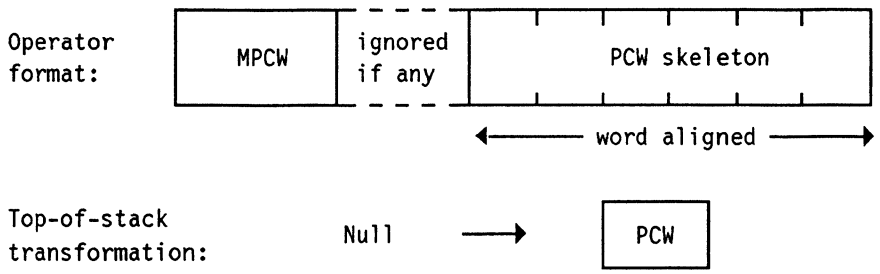
write	Copied from indicated DD
reindex	Enabled

The following service interrupts, which overlap those for reference chain evaluation, may be generated by INDX if the DD is exceptional:

Absent Segment	page table is absent
Binding Request	touched DD.element_size = 7
Touch Reference	<target> is untouched DD page DD is untouched DD

MPCW <make PCW>

MPCW is a literal operator that constructs a PCW at the top of the stack from a six-syllable parameter in the code stream. The parameter is taken from the first code word following the MPCW op code. Padding syllables, if any, from the op code to the end of the word containing the op code are ignored.



The valid ranges for PCW-skeleton values in user programs are:

skeleton_psi:	0 to 5
skeleton_pwi:	0 to length-1 of the code segment
skeleton-CS:	0
skeleton_LL:	sdll+1 to LL+1
sdll:	0 or 1
sdi:	1 to length-1 of activation record at level sdll

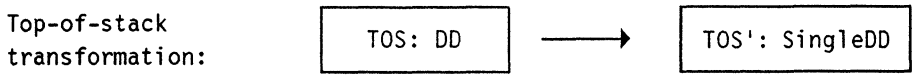
The contiguous group of fields skeleton_psi through skeleton_LL are copied from the skeleton into corresponding fields psi through lex_level of the PCW. The address couple (sdi,sdll) is evaluated to fetch a CSD; the ASD_ref field of the CSD is copied into the PCW. All other PCW fields are set to zero. The PCW tag is set to 7. A Touch CSD service interrupt is generated if the CSD is untouched.

Reference Modification Operators

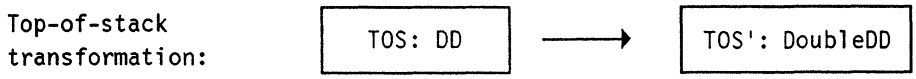
SEsz <set element_size ...>

These four operators set the element size of a copy DD:

SESW <set element_size to single words>

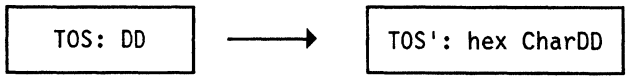


SEDW <set element_size to double words>



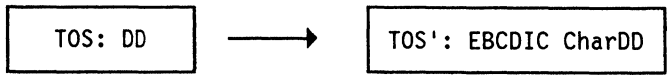
SE4C <set element_size to hex characters>

Top-of-stack transformation:



SE8C <set element_size to EBCDIC characters>

Top-of-stack transformation:



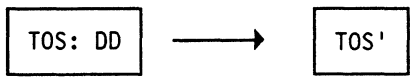
The argument must be an indexed or unindexed copy WordDD or CharDD; an IndexedDD must have reindex enabled. The result is the same DD with its element size set to a value implied by the operator name: single, double, hex, or EBCDIC for SESW, SEDW, SE4C, or SE8C, respectively. The index field of the DD is not modified.

If the argument is an IndexedWordDD, SE4C and SE8C require that the index value be less than 2**16. If the argument is an IndexedCharDD, all four operators require that the char_index be zero.

NOWR <disable write access>

The NOWR operator disables the write access right in references. The operator requires one argument, which must be a copy DD. The result is the argument with write access disabled.

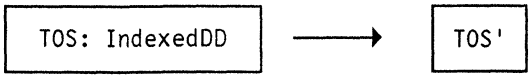
Top-of-stack transformation:



NORX <disable reindex>

The NORX operator disables reindexing in indexed DDs. The operator requires one argument, which must be an indexed DD. The result is the argument with reindex disabled.

Top-of-stack transformation:



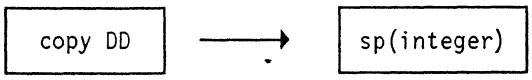
Descriptor Attribute Extraction Operators

The descriptor attribute extraction operators are provided to extract the value of certain attributes of DDs. In each case, a copy DD is consumed as the argument and a single_integer is left as the result.

GLEN <get length>

The argument must be a copy WordDD or CharDD; the result is the declared length of the designated segment, reported in the element_size of the DD.

Top-of-stack transformation:



The declared length is computed from three values in the ASD referenced by the DD:

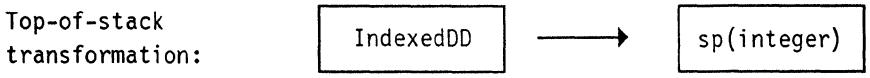
W = AS_length:	words in the designated actual segment
D = deficit:	words not counted in the declared length
N = nybbles:	4-bit characters reported in the declared length

Define S = W for unpaged DDs and S = W * 2**13 for paged DDs. Then the declared length L is defined for each element size as:

single:	L = S-D
double:	L = (S-D) DIV 2
EBCDIC:	L = (S-D)*6 + N DIV 2
hex:	L = (S-D)*12 + N

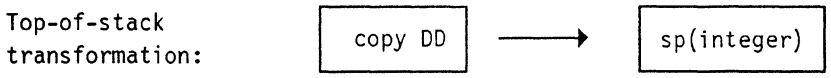
GINX <get index>

The GINX operator requires an IndexedWordDD or IndexedCharDD as its argument; the result is the virtual index of the DD. Note that for an IndexedDoubleDD, the virtual index is in single words.



GESZ <get element_size>

The GESZ operator requires a copy DD as its argument. The result is the element_size encoding of the DD, as a 3-bit integer; the result is 0, 1, 2, or 4 for a SingleDD, DoubleDD, hex CharDD, or EBCDIC CharDD, respectively and it is in {3, 5, 6, 7} if the DD had one of those invalid encodings. (GESZ is functionally equivalent to ISOL 42:3 on a copy DD.)



Read Evaluation Operators

aFOP <fetch operand value>

The common action aFOP is defined to fetch a single or double precision operand via an IRW or IndexedWordDD. The action can be described as a procedure with one formal parameter, the reference to be evaluated, and four possible results:

1. An operand value
2. A non-operand value (via an IRW or via IndexedSingleDD for LOAD)
3. A service interrupt
4. An error interrupt

For most clients, aFOP generates an error interrupt if the referenced word does not have a tag in {0,2}. The exceptions are value-call and LOAD, for which a non-operand may be a chained reference or a target, respectively. If the reference is an address couple or SIRW, or the reference is an IndexedSingleDD and the client is LOAD, the non-operand value is returned to the client operator.

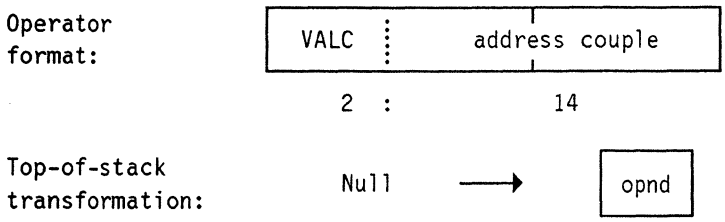
If the reference is an IRW and the referenced word has a tag of 0 or the reference is an IndexedSingleDD and the referenced word has a tag in {0,2}, the referenced word is fetched and the tag set to zero (if necessary) to form an operand value.

If the reference is an IRW and the fetched word has a tag of 2, its successor word (at the next higher memory address) is fetched. That word must have a tag in {0, 2}; the referenced word and its successor are joined as the first and second words of a double precision operand (with tag 2) to form an operand value.

If the reference is an IndexedDoubleDD, the referenced word and its successor are fetched; each must have a tag in {0,2}. The two words are joined as the first and second words of a double precision operand (with tag 2) to form an operand value. A split across a page boundary can occur if an IndexedDoubleDD is so constructed that its index refers to the last word of a page.

VALC <value call>

The VALC operator evaluates a reference chain whose beginning is an address-couple parameter; the target must be an operand, which is left on top of the stack. VALC is a two-syllable operator with a special structure, a 2-bit op code and a 14-bit variable-fence address couple:



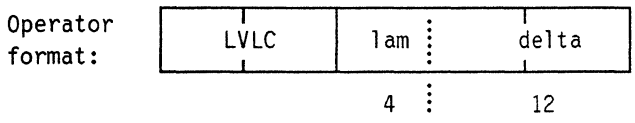
Reference chain evaluation performed by VALC is:

```
<Initial Reference>     ::=     {address couple}
<target>                ::=     {operand}

address couple          ------>  SIRW
                                 IndexedWordDD
                                 PCW
                                 <target>

SIRW                    ------>  SIRW
                                 IndexedWordDD
                                 PCW
                                 <target>

IndexedWordDD           ------>  <target>

PCW                     ------>  SIRW
                                 IndexedWordDD
                                 <target>
```

Reference evaluation is performed by invocation of the common action aFOP. An operand value is left on the stack as the result of VALC and any non-operand value is examined as an element of the reference chain. If a PCW must be evaluated, accidental entry is performed by invoking the common action aACCE. Return from the function causes VALC to be resumed in restart mode; the operator ignores its code parameter and consumes a stack argument as the target of the PCW.

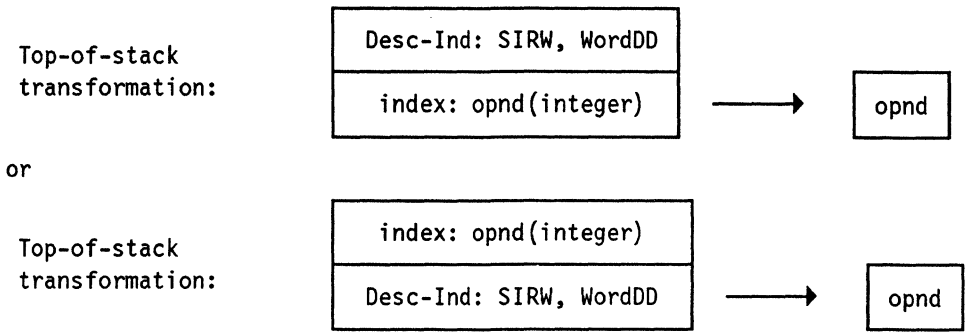
LVLC <long value call>

LVLC is equivalent to VALC, except the parameter is a fixed-fence rather than a variable-fence address couple. LVLC is a four-syllable operator whose appearance in the code stream is as follows:



NXLV <index and load value>

NXLV performs an INDX (index) operation to produce an IndexedWordDD and then evaluates the IndexedWordDD to fetch an operand. The required initial stack state is the same as that for INDX except that the DD is to be a WordDD.




```
<Descriptor Indication>    ::=    {copy unindexed word DD,
                                   IndexedWordDD,
                                   initial reference}

<Initial Reference>        ::=    SIRW
<Index target>             ::=    {unindexed WordDD,
                                   IndexedWordDD }

SIRW                       ⟶      SIRW
                                  <index target>
```

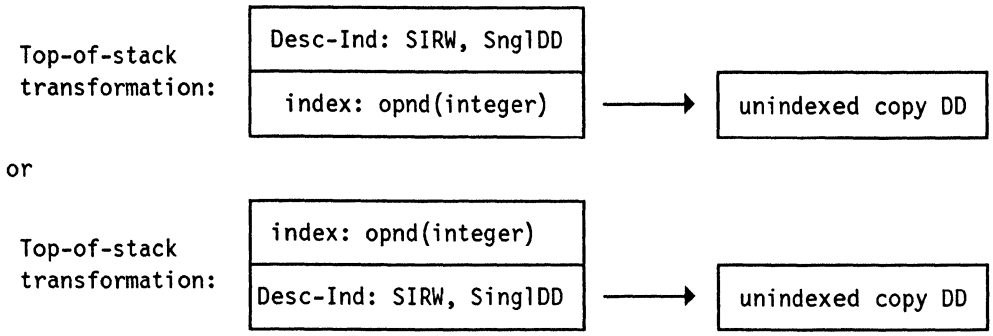
If the WordDD is successfully indexed, the ultimate target is fetched by read evaluation of the IndexedWordDD:

```
IndexedWordDD    ⟶    operand
```

The IndexedWordDD is evaluated by invoking the common action aFOP; an operand value is left on the stack as the result of NXLV.

NXLN <index and load name>

NXLN performs an INDX (index) operation to produce an IndexedSingleDD and then evaluates the IndexedSingleDD to fetch an unindexed DD; a copy of that DD is left on top of the stack. The required initial stack state is the same as that for INDX except that the descriptor indication or index target must be a SingleDD.




```
<Descriptor Indication>     ::=    {unindexed copy SingleDD,
                                    IndexedSingleDD,
                                    initial reference}

<Initial Reference>         ::=    SIRW
<index target>              ::=    {unindexed SingleDD
                                    IndexedSingleDD }

SIRW                        ⟶      SIRW
                                   <index target>
```

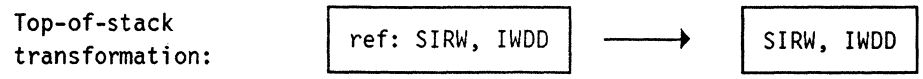
If the SingleDD is successfully indexed, the ultimate target is fetched by read evaluation of the IndexedSingleDD:

```
IndexedSingleDD  ⟶  unindexed DD
```

The target is fetched as a copy and left on the top of the stack. A service interrupt (Touch Load) is generated if the target of the IndexedSingleDD is untouched.

EVAL <evaluate>

The purpose of EVAL is to evaluate a reference chain in order to locate some target and then leave on top of the stack the reference whose evaluation produced the target.



Reference chain evaluation performed by EVAL is:

```
<Initial Reference>      ::=     {SIRW, IndexedWordDD}
<target>                 ::=     not {SIRW, IndexedWordDD, PCW}
SIRW                     ——→     SIRW
                                 IndexedWordDD
                                 PCW
                                 <target>
IndexedWordDD            ——→     * no evaluation -- see below *
PCW                      ——→     SIRW
                                 IndexedWordDD
```

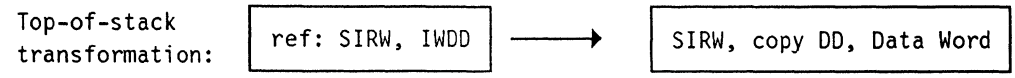
If a target is located, the reference whose evaluation produced the target is left on top of the stack as the result. If an IndexedWordDD is encountered, it is left as the result without being evaluated. In effect, an IndexedWordDD is treated as if it had been evaluated and a target had been the result.

If a PCW must be evaluated, accidental entry is performed by invoking the common action aACCE. EVAL deletes the Initial Reference and then invokes aACCE, so that the result of the accidental-entry procedure becomes the new Initial Reference.

Almost every data type is specified as either a valid reference or a target for EVAL. A stack-linkage word is not a proper target, but the processor cannot enforce this rule.

LOAD <load>

LOAD performs a single evaluation of the Initial Reference and if the result is a target, it is left on top of the stack.

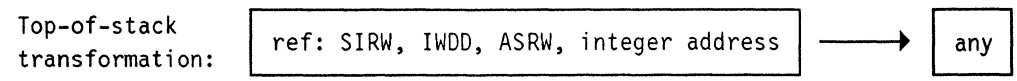



```
<Initial Reference>        ::=     {SIRW, IndexedWordDD}
<target>                   ::=     {operand, tag-4 word, tag-6 word,
                                    SIRW, any data descriptor}

SIRW                  ⟶   <target>
IndexedSingleDD       ⟶   <target>
IndexedDoubleDD       ⟶   operand (target)
```

Reference evaluation is performed by invocation of the common action aFOP. An operand value or other valid target is left on the stack as the result of LOAD. If the target is an original DD, it is fetched as a proper copy. A service interrupt (Touch Load) is generated if the target is an untouched DD.

LODT <load transparent>

LODT performs a single evaluation of the Initial Reference and leaves the result on top of the stack with no restriction placed on the type of the result:




```
<Initial Reference>     ::=    {SIRW, IndexedWordDD,
                                integer address (restricted),
                                ASRW (restricted) }
<target>                ::=    {any word}

SIRW                    ⟶      <target>
IndexedWordDD           ⟶      <target>
ASRW                    ⟶      <target>
Integer                 ⟶      <target>
```

The applicability of the LODT in user programs is partially restricted; the reference argument is never an operand or ASRW and the target is never a touched original DD.

If the Initial Reference is an SIRW, IndexedWordDD, or ASRW, it is evaluated normally; a single-word fetch is performed regardless of the element_size of an IndexedWordDD. An integer Initial Reference is interpreted as an absolute address from which the target is fetched.

The addressed target word is left on top of the stack. If its tag is 2, the second word of the item left on top of the stack is zero. (Apart from the handling of tag=2, the LODT operator does not examine or alter the target word in any way.)

Normal Store Operators

aSOP <store operand value>

The common action aSOP is defined to store a single or double precision operand value via an IRW or IndexedWordDD. The action can be described as a procedure with two formal parameters, the reference to be evaluated and the object stored. An IndexedWordDD must be write-enabled.

The type of store object must match that of the target location and one of the following statements must be true:

```
The operand is single precision and:
   the final reference is an IndexedSingleDD or
   the final reference is an IRW and the target is in {single datum}.

The operand is double precision and:
    the final reference is an IndexedDoubleDD or
    the final reference is an IRW and the target is in {double datum}.

The set {single datum} is defined as the type union:
    {tag-0 word (single precision operand),
     tag-4 word,
     tag-6 word (uninitialized single datum) }.

The set {double datum} is defined as the type union:
    {tag-2 word (double precision operand) }.
```

If the operand is single precision, it is stored at the target location.

If the operand is double precision, the first word is stored at the target location and the second at the successor of the target. The successor word must contain a Data Word. A split across a page boundary occurs if the final reference is an IndexedDoubleDD so constructed that its index refers to the last word of a page.

Single precision operands are always stored with tag = 0 and both halves of a double precision operand are stored with tag = 2. Note that the target type can be changed by a normal store operation; an uninitialized datum can be changed to an operand and a single or double precision operand can be changed to the other if store is via an IndexedWordDD.

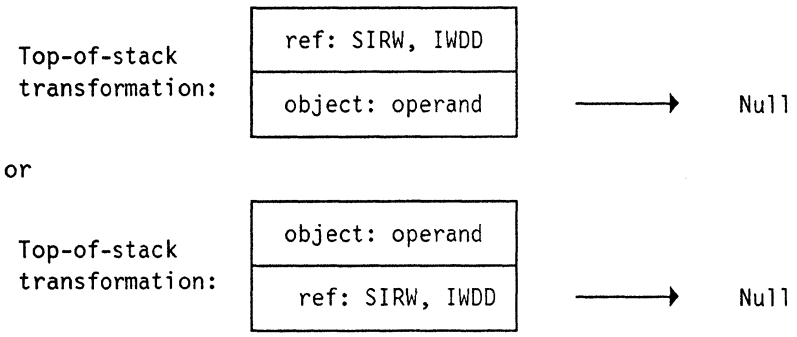
If an error interrupt is generated while the second half of a double precision value is being stored, the first half of the item has not been stored.

STOD <store delete>
STON <store non-delete>

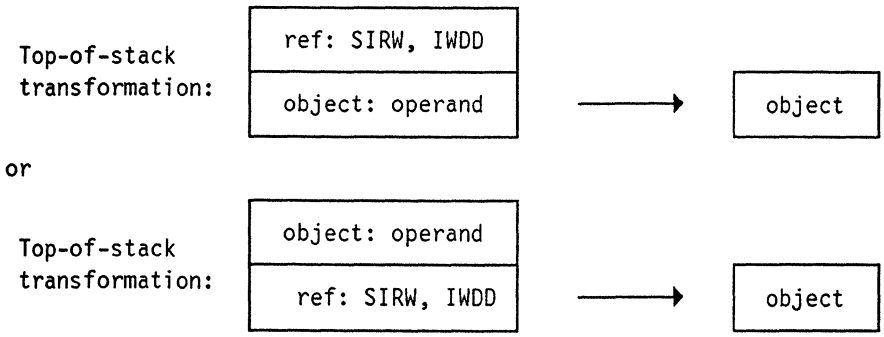
Normal store operators evaluate a reference chain in order to store an operand from the stack (the store object) into a target location. Reference chain evaluation performed by store operators is:

```
<Initial Reference>     ::=     {SIRW, IndexedWordDD}
<target>                ::=     Data Word

SIRW                    ------>  SIRW
                                 IndexedWordDD
                                 PCW
                                 <target>

IndexedWordDD           ------>  <target>

PCW                     ------>  SIRW
                                 IndexedWordDD
```

For STOD and STON, the Initial Reference and the operand are required on top of the stack, in either order. (For best performance, address couple cases should emit NAMC STO-; other cases should place the reference is TOS2.) STOD deletes both the Initial Reference and the operand from the stack:



STON deletes the Initial Reference but leaves the operand unchanged on top of the stack:



The operator performs read evaluation of the reference chain followed by store evaluation of the final reference to the target. If a PCW must be evaluated, accidental entry is performed by invoking the common action aACCE, so that the result of the accidental entry procedure becomes a new Initial Reference.

The operand is stored at the final target location by invocation of the aSOP common action.

Overwrite Operators

Overwrite operators perform a single evaluation of the Initial Reference and store some item from the stack (the store object) into the resultant target location. The arrow indicates store evaluation for these operators, although under some circumstances, the processor may also read the target contents.

```
<Initial Reference>      ::=   {SIRW, IndexedSingleDD,
                                ASRW (restricted) }
<target>                 ::=   any item
SIRW                     ⟶     <target>
IndexedSingleDD          ⟶     <target>
ASRW                     ⟶     <target>
```

The topmost stack item must be an Initial Reference; the second stack item is the Store object. The store object may be of any type.

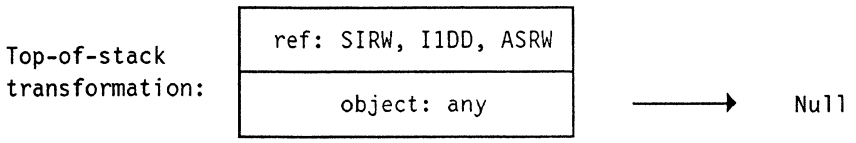
A write evaluation of the initial reference is performed; write access is required.

The applicability of overwrite operators in user programs is partially restricted; the ASRW form of reference is not used, the store object is typically a reference, operand, or tag-4 or tag-6 word and the store target is not an original descriptor, stack-linkage word, and so forth. Software requires that references generally not be stored in segments or activation records with longer lifetime than the referent.

Overwrite operators store single words; OVRD and OVRN are oblivious to double precision. If the store object is double precision, only the first word is stored, with its tag of two. If the reference addresses a tag-2 word, only that word is overwritten; its successor word is unchanged. Note that each case can produce an unpaired double precision word in memory.

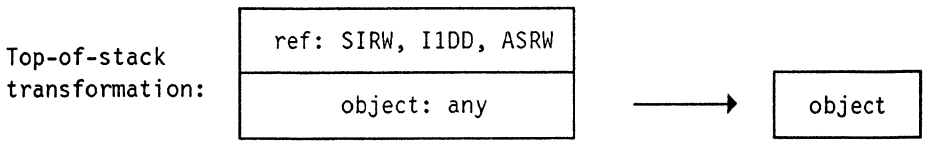
OVRD <overwrite delete>

An overwrite operation is performed. Both the Initial Reference and the store object are deleted from the stack.



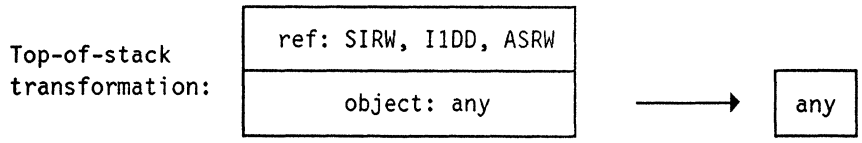
OVRN <overwrite non-delete>

An overwrite operation is performed. The store object is left unchanged on top of the stack and the Initial Reference is deleted.



RDLK <read and lock>

RDLK is identical to OVRD, except that the word previously occupying the target location is left as the stack result.



RDLK is a synchronization primitive. The following requirements must be met by all operating systems and by any other processing elements (such as I/O or communications processors) that share the system memory and use the RDLK convention for synchronization:

1. RDLK reads the former contents and writes the new contents of the referenced word as an indivisible operation; no other processor can access the referenced word between the read and write steps of a RDLK operator.
2. Store operations for each processor effectively occur in order. That is, if a code stream on one processor initiates stores into location A and then into location B, any processor that finds the new value at B (with RDLK) and subsequently examines A must find the new value at A.
3. No data-fetch memory operation that follows the RDLK in the dynamic code stream of a processor may take place until after the RDLK operation.

Statements 2 and 3 restrict the ability of a system to reorder the manipulation of independent pieces of the system, such as the contents of different memory locations.

Interlock Operations

An interlock is a data object used to provide mutual exclusion between critical regions of program code. Through a programmer-designed convention, an interlock can be associated with any desired program state. By *locking* that interlock prior to entering a critical region and *unlocking* it upon leaving the region, a process can be guaranteed exclusive access to the same associated state, assuming that all other processes observe the same locking convention.

Five operations on interlocks are defined:

Lock
Unlock
Status
Break
Arrogate

In addition, operating-system software may intervene to abort an attempt by a process to Lock an interlock, as when the process is itself aborted.

Because they serve as synchronization (mutual exclusion) primitives, the interlock operations are subject to the same operator-ordering constraints as defined for the RDLK operator. Furthermore, the operations are atomic in that all the state associated with the interlock changes at once, in the perception of the program. (Interlock operations differ from RDLK in that they release the waiting processor to other processes; when RDLK is used for locking, the program must either retain the processor in a "busy wait" or take explicit action to relinquish it.)

An interlock may exist in any of four stable states:

Free (the initial and resting state)
Locked_Uncontended
Locked_Contended (one or more processes waiting to Lock)
Broken

Additional states associated with a non-Free interlock include its owner (the stack_number of the process that successfully completed a Lock, Break, or Arrogate operation) and, for the Locked_Contended state, a list of contenders (processes waiting to Lock).

The A16 uses the "basic" approach to interlock implementation: uncontended locking and unlocking are performed by unassisted operators, but these operators generate service interrupts to invoke software when contention must be resolved.

The complete implementation of the interlock operations is provided in part by the operators and in part by the software. Therefore, operations must be defined at a rather higher level of abstraction than is characteristic of the rest of the specification. In particular, some multiprogramming and multiprocessing terminology must be introduced without rigorous definition.

Contention for a lock is resolved by placing the contending process into a *wait* state and freeing the processor to move to another process (stack). When the lock is later unlocked, one contending process is granted ownership and placed in *ready* state; when a processor is available, it is associated with a process chosen from a list of ready processes. If there are several processes contending for one lock, they must succeed to ownership of the lock one at a time, in some order. When there are more ready processes than processors, there must be some mechanism to allocate the processors among the processes. Both the wait ordering and the processor allocation are based on a concept of *priority*, where higher priority denotes greater urgency of the process.

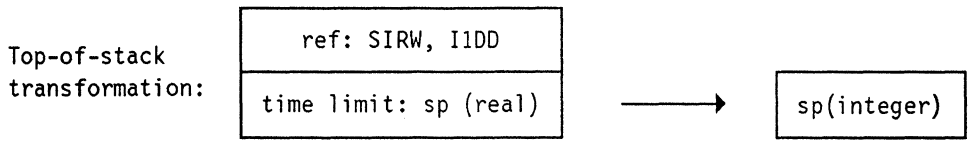
Each interlock operation requires an Initial Reference that directly references the interlock:

```
<Initial Reference>        ::=   {SIRW, IndexedSingleDD}
<target>                   ::=   interlock

SIRW                       ——→   <target>

IndexedSingleDD            ——→   <target>
```

The LOK and UNLK operators require the topmost stack item to be an initial reference with write access enabled; the addressed target must have an appropriate tag. Equivalent checking of the reference parameter must be performed by or on behalf of the procedure invoked to perform the Status, Arrogate, and Break operations.

LOK <lock interlock>

The Lock operation is initiated by the LOK operator, which has two arguments: the Initial Reference and a time limit.



The time-limit argument is a single-precision floating-point number; its value specifies the maximum duration in seconds of an attempt to Lock an interlock. If limit is less than zero or limit is greater than 86400 (one day), the wait is indefinite; if limit equals zero there is no wait.

LOK produces one single_integer result, where 1 indicates success and an even value indicates an exception condition:

0: abort (the interlock operation was aborted by operating system)
2: timeout (the time limit elapsed)
4: broken (the interlock is in Broken state)

If the interlock is in Free state, it becomes Locked_Uncontended and the operator result is 1 (success).

If the interlock state is Broken, its state is unchanged and the operator completes with a result of 4.

If the interlock state is Locked_Uncontended or Locked_Contended and the time limit is zero, the operation completes and the result is 2.

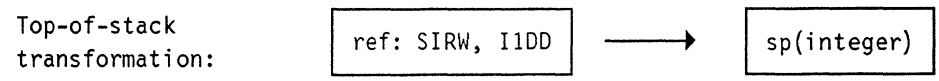
If the interlock state is Locked_Uncontended or Locked_Contended and the time limit is nonzero, the interlock becomes Locked_Contended and the locking process is placed into a waiting state and recorded as a contender for the interlock. The priority of the process that owns the interlock is made at least as great as that of any process that is contending for it. The processor is released to another ready process. The waiting process will be deleted from the list of contenders, changed from waiting to ready state, and the operation will be completed when any of the following events occurs:

- An Unlock operation on the interlock causes this process to become the new owner; the result is 1 (success).
- The time-limit interval elapses; the result is 2.
- A Break operation is performed on the interlock; the result is 4.
- The operating system causes the operation to be aborted; the result is 0.

UNLK <unlock interlock>

The Unlock operation is initiated by the UNLK operator, which has one argument, the Initial Reference. UNLK produces one single_integer result, where 1 indicates success and an even value indicates an exception condition:

4: interlock state was Broken
6: interlock state was Free



If the interlock state is Free or Broken, the state remains unchanged and the result is 6 or 4, respectively.

If the interlock state is Locked_Uncontended, its state is changed to Free; the result is 1.

If the interlock state in Locked_Contended, the following actions occur and the result is 1:

- Any necessary priority readjustment is performed on the process previously recorded as owner of the interlock.
- The highest-priority process among those contending for the interlock is recorded as the new owner and changed from waiting to ready state; the Lock operation initiated in that process completes successfully.
- If the new owner was the only contender, the state of the interlock is changed to Locked_Uncontended; otherwise the state remains Locked_Contended and the lower-priority contenders remain in the wait list.

Status Interlock

The interlock status function is performed by a procedure with one parameter, the Initial Reference. The function returns a single-precision value containing the following information:

```
[47:24]  owner stack_number; 0 if none
[23:22]  undefined
[ 1: 2]  interlock state:
                0: Free
                1: Locked_Uncontended
                2: Locked_Contended
                3: Broken
```

Break Interlock

The Break operation is performed by a procedure with one parameter, the Initial Reference; the function returns a single-precision result containing the Status value prior to the Break operation. The interlock is set to Broken state, the owner is set to the value of SNR, and any contending processes are changed from waiting to ready state (with the Lock operation returning four).

Arrogate Interlock

The Arrogate operation is performed by a procedure with one parameter, the Initial Reference; the function returns a single-precision result containing the Status value prior to the Arrogate operation. The owner is set to SNR and the state is set to Locked_Uncontended, unless it was already Locked_Contended. In this case the state remains unchanged, the list of waiting contenders is retained, and the priority of the arrogating process is set at least as high as that of any contender.

Processor State Operators

This section describes operators that interact with processor state, primarily the state of the currently executing code stream and the state of the stack in which the processor is running.

Code Stream Pointer Distribution

The processor code stream pointer is initialized by the distribution of PCW or RCW code stream pointer components.

1. The PSN (Program Segment Number) is set from the ASD_number field in the PCW or RCW.
2. The PWI and PSI are set from the corresponding fields in the PCW or RCW. PWI must be in the range {0 to L-1}, where L is the length of the code segment; PSI must be in the range {0 to 5}.

Branching Operators

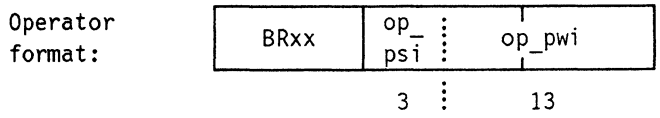
Branching operators provide for altering the processor's code stream pointer component. They may change the point of execution in the current code segment or establish a new code segment with an initial point of execution.

Branches may be conditional or unconditional. Conditional branches alter the code stream pointer or continue sequential execution depending upon the Boolean interpretation of a stack argument.

Branching operators are classified as static or dynamic branches and are discussed in the following paragraphs.

Static Branches

Static branches always point within the current code segment. That point is indicated by a two-syllable parameter:



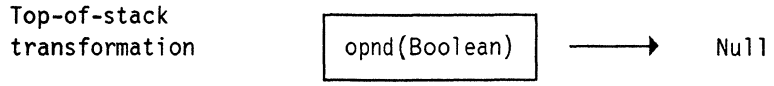
The high-order three bits of the parameter are interpreted as the new PSI value and the low-order 13 bits as the new PWI value. The maximum valid op_pwi is one less than the length of the current code segment; the maximum valid op_psi is five.

BRUN <branch unconditional>

Processor registers PSI and PWI are set from the parameter and the processor is conditioned to execute next the operator at that point in the current code segment.

BRTR <branch true>
BRFL <branch false>

The top-of-stack argument must be an operand; it is interpreted as a Boolean value. If BRTR finds it to be True or BRFL finds it to be False, processor registers PSI and PWI are set from the parameter and the processor is conditioned to execute next the operator at that point in the current code segment. Otherwise, sequential execution continues.



Dynamic Branches

Dynamic branches take their code stream pointer values from a branch destination item on top of the stack. They may branch to a computed point within the current code segment or to a point in an arbitrary code segment.

```
<Branch Destination>      ::=    {operand, PCW, initial reference}
<Initial Reference>       ::=    {SIRW}

<target>                  ::=    PCW

SIRW                      ⟶      <target>
```

Branching within the current code segment is indicated if the branch destination item is an operand. It is integerized with rounding (RAI function), if required, to produce a 14-bit integer, which is interpreted as the code segment index in units of half-words (three syllables):

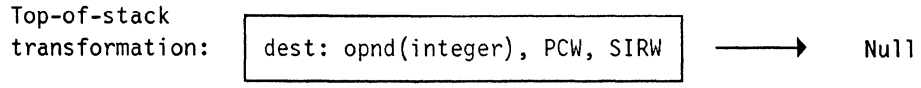
```
dyn_pwi          [13:13]     New PWI value
dyn_half_word    [ 0: 1]     The new PSI value is dyn_half_word * 3
```

The largest valid dyn_pwi is one less than the length if the current code segment.

Branching to a point in an arbitrary code segment is indicated if the branch destination item is a PCW or an SIRW to a PCW. PCW.lex_level must match the current value of LL. The PCW code stream pointer is distributed as specified under "Code Stream Pointer Distribution" in this section.

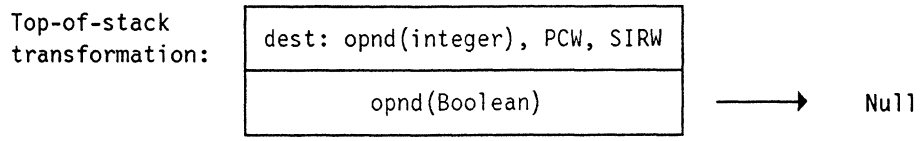
DBUN <dynamic branch unconditional>

A branch destination argument is required on top of the stack.



DBTR <dynamic branch true>
DBFL <dynamic branch false>

The branch destination argument is required on top of the stack. The argument second from top-of-stack must be an operand; it is interpreted as a Boolean value. If DBTR finds it to be True or DBFL finds it to be False, a branch is executed using the top of the stack branch destination item exactly as in DBUN. Otherwise, sequential execution continues.



Stack Frame Operators

Operators in this group provide for procedure entry and exit and the creation and destruction of stack frames.

Procedure Entry Operators

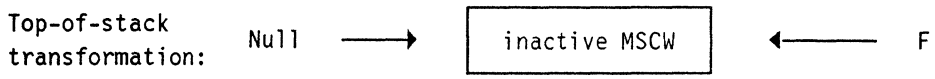
The sequence of operations in normal procedure invocation has already been outlined:

1. Perform a Mark Stack operation
2. Place a reference to the code onto the stack
3. Place any parameters onto the stack
4. Perform an Enter operation
5. Place objects onto the stack to extend the new AR

MKST <mark stack>

Mark Stack operators *mark the stack* for the beginning of a new stack frame by creating at the top of the expression stack an incipient activation record linked into the historical chain.

MKST builds an inactive MSCW on top of the stack and links it at the head of the historical chain.



The displacement between the new MSCW location and the base of the current stack must be in the range (1 to 2**16-1). The displacement between the new MSCW and the previous one must be in the range (1 to 2**13-1). (When the stack frame is too large to be spanned by a history link, a Frame Overflow condition exists, which is normally detected by the subsequent Enter operation.)

F is set to point at the new MSCW on the stack.

MKSN <mark-stack bound to name-call>

This operator is functionally equivalent to the MKST operator and is subject to the following rules:

1. The immediate next operator in the code stream must be NAMC (name call).
2. The address couple in the NAMC must be the initial reference for the corresponding ENTR.

3. With two exceptions, the addressing environment, reference chain, and code segment addressing must remain the same when the corresponding ENTR is executed as when the MKSN was executed. *Code Segment Addressing* refers to the components of the PCW and ASD that are used to address the code segment; the term also includes the status (untouched, absent, present) of the segment. The exceptions are:

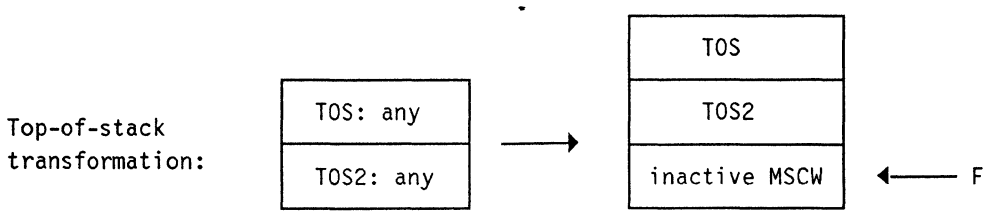
   a. The reference chain does not yield a PCW. As a result of the ensuing interrupt, the reference chain, PCW, and code-segment addressing are subject to modification.

   b. Another mark-stack operation occurs, either explicitly or via an interrupt. In this case, code segment status (untouched, absent, or present at a specific address) is subject to change, but the addressing environment, reference chain, and PCW must not change.

4. Between the execution of the MKSN and the subsequent execution of the corresponding ENTR, the stack linkage remains valid and *inactive*, however, other fields in the MSCW and the entire SIRW/PCW location are undefined.

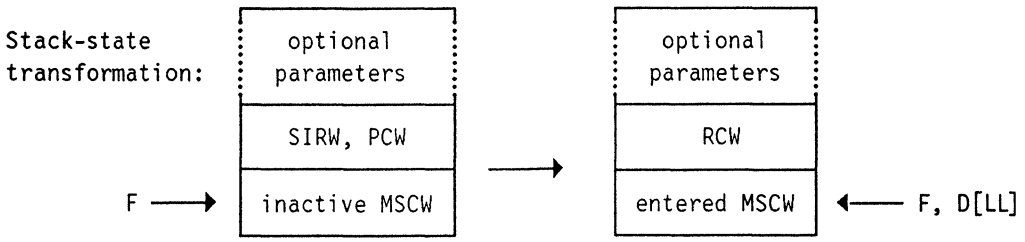
IMKS <insert mark stack>

IMKS creates an incipient activation record exactly as does MKST (mark stack), except that the new MSCW is inserted beneath the two top of stack items. There must be at least two items in the expression stack at the start of an IMKS operation.



ENTR <enter>

The initial stack state for ENTR assumes prior execution of a Mark Stack operator and placement of a procedure reference onto the stack. S > F > D[LL]+1 must be true, an inactive MSCW is required at the F stack location, and an Initial Reference is required at the F+1 stack location.



```
<Initial Reference>      ::=      {SIRW,
                                   <target> }
<target>                 ::=      {PCW}

SIRW          ——→        SIRW
                         <target>
```

With a PCW, ENTR consists of the following functional tasks:

1. Complete the MSCW, inserting it at the head of the appropriate environmental chain:
   a. ENTR forms an Environmental Link to address the base of the global AR; this Environmental Link is inserted into the MSCW to complete the environmental chain that defines the addressing environment of the new procedure. (Note that PCW.lex_level must be in the range {1 to LL+1}.)
   b. The global AR is identified by the form of the procedure reference. If the PCW is the initial reference or, for the A16, if the PCW is directly referenced by the address couple in a NAMC or LNMC immediately following the most recent MKST or MKSN, the global AR is the one at lex level L = PCW.lex_level-1 in the initial addressing environment. In these cases, an Environment Link to the global AR is constructed by implicit invocation of the NAMC operator with the address couple (L,0). Otherwise, the final reference to the PCW is an SIRW, whose Environment Link points directly to the global AR.
   c. The word at the base of the global AR must be an entered MSCW.
2. Preserve the code stream pointer and related state in an RCW:
   a. When entry occurs from a code segment, whether by explicit invocation of ENTR or by accidental entry or interrupt, the operation builds an RCW containing the processor code stream pointer and the values of LL, the CS (control state) Booleans, and the Boolean accumulators. For explicit ENTR, the RCW code stream pointer designates the syllable following ENTR. For accidental entry (ENTR invoked by aACCE), the code stream pointer designates the operator invoking aACCE. For interrupt entry, the code stream pointer is determined by the specific interrupt situation.
   b. The RCW is stored in place of the Initial Reference.

3. Initialize processor state for the new procedure:

   LL is set from PCW.lex_level and D[LL] is set from F to address the base of the activation record. The processor CS Boolean is set from PCW.control_state.

   Display registers are updated to reflect the new addressing environment. For the A16, if the new PCW reference was the address couple in a NAMC or LNMC immediately following the most recent MKST or MKSN, or the Initial Reference was a PCW, no update is necessary, since the global AR of the new activation record is already in the addressing environment. Otherwise, display update begins at D[LL-1], the new global AR.

   The expression stack is appended to the new activation record, making any procedure parameters accessible.

   The processor code stream pointer state is initialized from the PCW.

aACCE <accidental entry>

The common action aACCE is invoked by some reference-chain evaluation operators to evaluate a PCW. The action is automatic invocation of the parameterless procedure (function) defined by the PCW; it is defined as three steps:

1. Invoke MKST
2. Place the reference to the PCW onto the stack
3. Invoke ENTR

The accidental entry and subsequent return leave the processor executing the operator that initiated the accidental entry. (Unlike explicit ENTR, which builds an RCW referencing the successor operator, aACCE points the RCW at the invoking operator.) The invoking operator provides the value for rs. In particular, value-call operators set rs to 1 to cause reentry in restart state.

Procedure Exit Operators

There are three operators for deleting a stack frame and reinstating the prior topmost stack frame and its code stream.

EXIT	terminates a subroutine and leaves no result
RETN <return>	terminates a function and leaves the top of stack item as a result
RTN2 <return two items>	terminates a procedure that leaves two top-of-stack results

These operators perform the EXIT operation, which deletes the topmost stack frame from the stack and restores processor state for the prior topmost stack frame. The base location of the topmost activation record is addressed by D[LL] and the prior topmost AR is located as the first entered AR below D[LL].

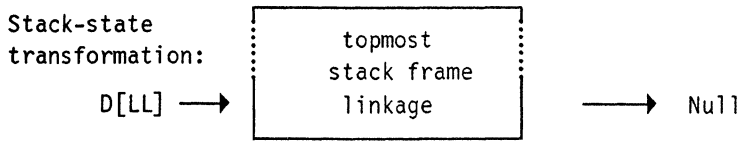
If the stack linkage of the topmost activation record has the block_exit bit set, the Exit operation is not performed; a Block Exit service interrupt is generated instead.

The topmost stack frame is deleted from the stack by setting the top of stack pointer at the first word below the stack linkage. At the start of the operation, S > D[LL] must be true (not counting any return arguments). The F register is set to point at the first MSCW on the historical chain headed by the activation record being exited. LL is set from the value saved in the RCW. D[LL] is reset to address the base of the prior topmost activation record. If F designates an incipient AR, the historical chain is traversed until an entered AR is found. The CS (control state) Boolean and the Boolean accumulators are restored by distributing values saved in the RCW of the original stack frame. Display registers are updated to reflect the new addressing environment; the environmental chain is traversed beginning at D[LL].

The processor code stream pointer is initialized from the RCW. If rs is set, the processor is conditioned to execute in restart mode the operator addressed by the new codes stream pointer.

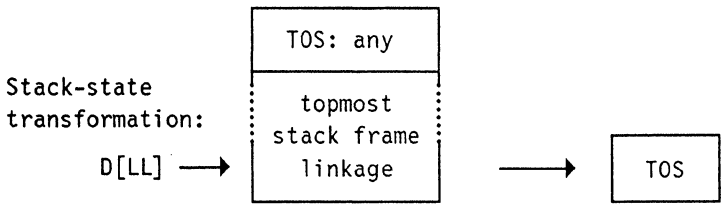
EXIT <exit>

The EXIT operator performs an Exit operation for a procedure that returns no value on the stack.



RETN <return>

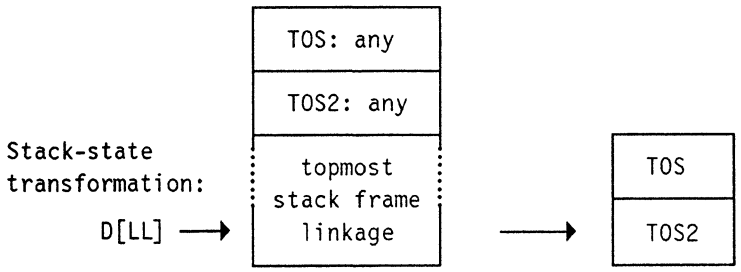
RETN performs an Exit operation for a procedure that returns a result on the stack. It retains the initial top-of-stack item and pushes it back onto the expression stack after the topmost stack frame is deleted.



RETN requires an item from the expression stack. In user programs, the item should be an operand or a reference to some object that is global to the procedure being exited. Trusted software must prevent the return of a copy DD referencing a segment being deallocated by the exiting of this block.

RTN2 <return two items>

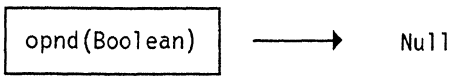
RTN2 performs an Exit operation for a procedure that returns two results on the stack. The topmost two items in the topmost stack frame are placed in the same order at the top of the resumed stack frame. RTN2 requires two items from the expression stack.



Assertion Operators

The assertion operators, ASRT and AVER, require and consume one operand on the stack. The argument is interpreted as a Boolean value. If it is False, a False Assertion interrupt is generated; the argument value is available to software for analysis. If the code stream is resumed by exiting from the interrupt procedure, the argument is gone from the stack and the subsequent operator is invoked.

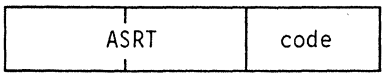
Top-of-stack transformation:



ASRT <assert>

The ASRT operator performs an assertion operation. It has a one-syllable code parameter, which is available to software if a False Assertion interrupt is generated.

Operator format:



AVER <aver>

The AVER operator performs an assertion operation; it has no code stream parameter. (The verb *aver* is defined in *Webster's New Collegiate Dictionary* as "to verify or prove to be true..., to declare positively"; it is not an acronym or abbreviation.)

Top-Of-Stack Operators

These operators delete, duplicate, or reorder top-of-stack items. There is no restriction on the type of stack item that will be acted upon, however, as operator arguments, the items must be in the expression stack to avoid Stack Underflow interrupts. Note that an item may comprise either one or two words.

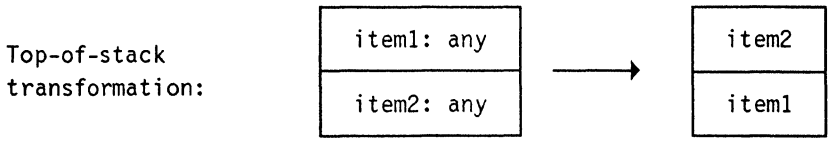
DLET <delete top of stack>

DLET requires one item on top of the stack and deletes it from the stack.

Top-of-stack transformation:

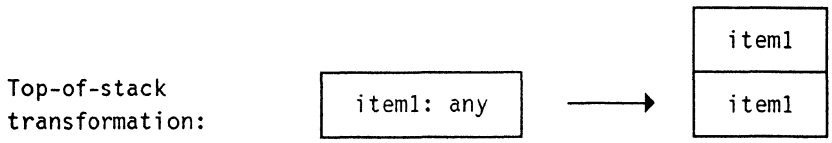
EXCH <exchange top of stack>

EXCH requires two items on top of the stack and interchanges their order in the stack.



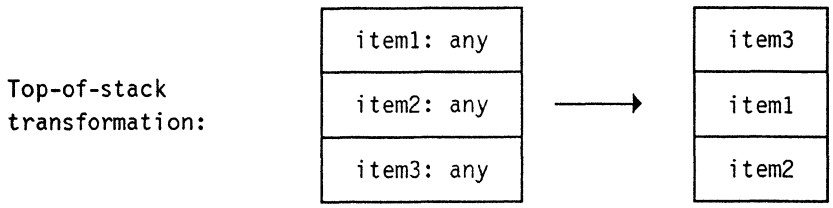
DUPL <duplicate top of stack>

DUPL requires one item on top of the stack, creates an exact duplicate of it, and leaves both items on top of the stack.



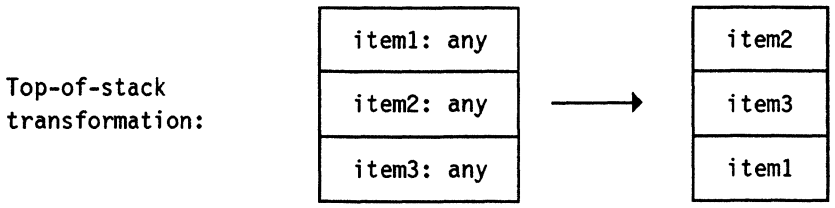
RSUP <rotate stack up>

RSUP requires three top of stack items. The third from top item is rotated up to become the top of stack item.



RSDN <rotate stack down>

RSDN requires three top of stack items. The top item is rotated down to become the third from top of stack item.



Processor-State Manipulation Operators

Operators in this group directly affect various elements of processor state, often reading state to the top of stack or setting state from the top of stack.

DEXI <disable external interrupts>

DEXI conditions the processor to ignore all external interrupts and sets the processor CS Boolean to one (control state). The stack is unaffected.

DEXI is partially restricted; DEXI and EEXI may be safely used only around short segments of code that must be executed atomically and are of short duration.

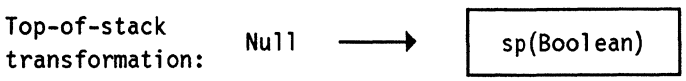
EEXI <enable external interrupts>

EEXI conditions the processor to respond to external interrupts and resets the processor CS Boolean to 0 (normal state). If any external interrupt is pending when EEXI is executed (in control state), the interrupt occurs immediately following the EEXI operator, even if the immediate successor operator is DEXI.

EEXI is partially restricted; it is used only as a companion to DEXI.

ROFF <read and reset overflow flip-flop>

ROFF leaves on top of the stack the Boolean value of OFFF (overflow flip-flop) and then unconditionally resets OFFF to false.

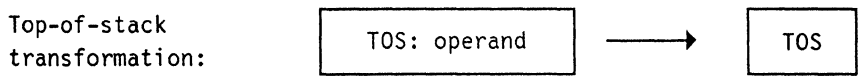


RTFF <read true-false flip-flop>

RTFF leaves on top of the stack the value of TFFF as a Boolean operand, True or False. The top of stack transformation is the same as for ROFF.

SXSN <set external sign flip-flop>

SXSN requires an operand on top of the stack. SXSN sets EXTF (EXTernal sign Flip-flop) to the value of bit 46 of the top of stack item. The operand is left unchanged on the stack.



RTOD <read time of day clock>

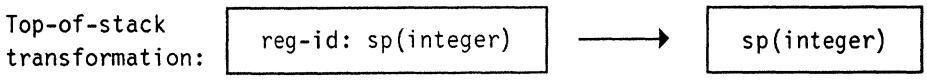
The restricted operator RTOD leaves on top of the stack a 36-bit integer containing the value of the time of day clock. The range of values is {0 to 2**36-1} in units of 2.4 microseconds, but is subject to change in later A Series systems.

RSNR <read SNR>

RSNR leaves on top of the stack a single_integer containing the SNR value.

RPRR <read processor register>

RPRR requires one 6-bit integer stack argument, which is interpreted as a processor register identification (register Id). The result left on top of the stack is the value of the specified register. This result value is a k-bit integer, where k is the width of the target register.



Readable processor registers are associated with register ids that are a subset of integers in the range (0 to 63); the valid subset, along with the register width, is shown in the following table:

register ID	register name	width
LL	D[LL]	16
37	BOSR	0
38	F	16
52	S	16
53	SNR	15

D[LL], F, and S report displacements relative to the base of the current stack; BOSR reports 0.

The value reported for the S register is the displacement of the top-of-stack word after the RPRR argument has been consumed.

The RPRR operator is partially restricted; user code should access only SNR (register 53). (Note that this use is obsolescent, superseded by the RSNR operator.) The various pointer registers should be used only as stack-relative differences, as in S-F, F-BOSR, and D[LL]-BOSR.

Data Array Operators

Operators in this group perform functions on arrays specified by data descriptor stack arguments. The functions applied generally consist of sequential processing of one or more arrays of word or character elements. Termination occurs when an element length has been exhausted or when some condition is satisfied.

Data in one of the argument arrays may be modified, the arrays may be processed in order to produce a result, or both actions may occur. Results are indicated by items left on top of the stack or by the setting of one or more processor Boolean accumulators.

Array operators typically accept IndexedDDs as arguments. For an operator to access the data, the actual segment must be present.

Paged Array Operations

In general, arrays may be paged segments. The exception is that edit tables must be unpaged. (I/O subsystems, which are not specified here, typically require unpaged buffers.)

A service interrupt (Touch Reference or Absent Data) occurs when an attempted page switch encounters an untouched or absent page. (Error interrupts can also occur at a page boundary if the ASD or page table structure is incorrect.) A characteristic of the Data Array operators is that a service interrupt can occur in mid-operation; such an occurrence is generically called a *page boundary interrupt*, a term that also includes Source and Destination Boundary interrupts. After the service has been performed (necessary segments are present), the operation is resumed. In general, the processor takes up where it stopped, although some operators can be simply repeated from the beginning.

Except for the cases previously noted, Data Array operators automatically attempt to switch to the next page when an actual segment is exhausted.

It should be noted that a reference to a page boundary can take one of two forms: a pointer to the last+1 element (index = 2**13 for the A16) of the lower page or a pointer to the first element (index = 0) of the upper page.

It is convenient to formalize all adjustment of descriptor index values as using aAPIX to apply a suitable increment; the common action switches pages as needed.

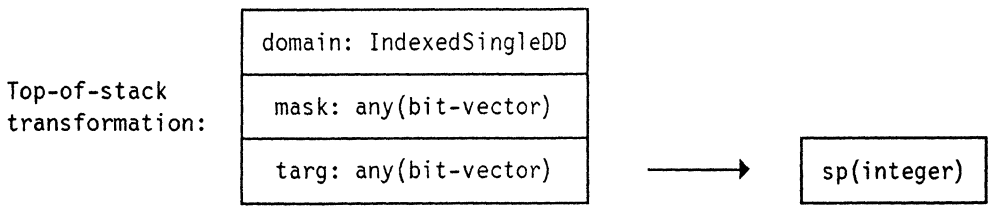
Searching Operators

There are two searching operators: SRCH (masked search for equal) and BMS (bounded masked search for equal). Each search an array backwards for the first word that is bitwise equal to a target value after both the word and target value have been masked.

SRCH <masked search for equal>

SRCH scans an array called the *domain*, which is a virtual segment. The array is searched from an indexed starting point back towards the base for a word that matches a target value in selected bits. Any or all of the 52 bits (word and tag) may be matched. The result is a single_integer that is the virtual index of the matching word or -1 if the search fails.

SRCH requires three arguments on top of the stack: the search domain, the mask, and the target value.



The domain argument must be an IndexedSingleDD; reindex must be enabled.

Both the mask and target value are bit vectors of length 52. The target value is logically ANDed with the mask before it is used for comparison. The second word of any double precision target value or mask is ignored.

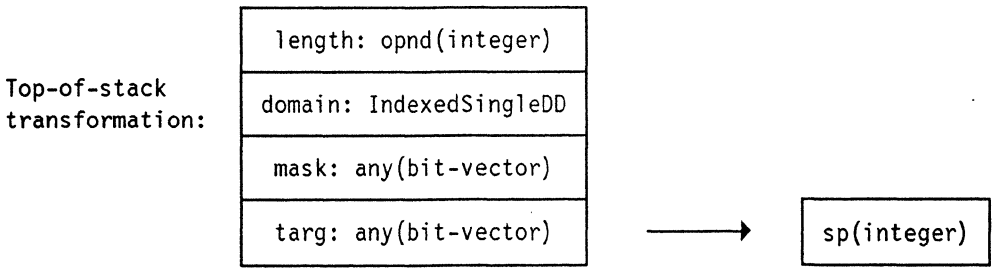
The word referenced by the IndexedSingleDD is logically ANDed with the mask and compared to the (masked) target value. If a matching word is found, its index (relative to the virtual segment) is left on the stack as the SRCH result. If there is no match and the index is nonzero, the index is decremented by one and the search continues. If there is no match and the index is zero, the segment is exhausted and the failure result is returned.

Note that on pre-ASD A Series systems, the segment described in the previous paragraph is the actual segment; only one page is searched if the original domain argument designates a paged array.

BMS <bounded masked search for equal>

BMS is restricted in code files that can run on other systems, because it is not implemented on all older A Series systems.

BMS is a variation of SRCH, with an additional argument that provides an upper bound on the length (number of words) of the search. The length argument appears topmost on the stack:



The length must be an operand. It is integerized with the RaI function, if necessary; its integer magnitude must not exceed MaxLen.

The domain, mask, and target arguments; the failure value; and the search algorithm are as defined for SRCH with the exception of the following:

- The operator terminates and returns the failure result if the length is zero or negative at the beginning of any iteration (including the first)
- The length is decremented after each word is examined
- It is an error if the virtual segment is exhausted before the length is decremented to zero

Pointer Operators

Pointer operators handle sequences of word or character elements in data arrays. There are pointer operators for scanning, transferring, comparing, and editing the element sequences in various methods. Some pointer operators handle only a source or a destination sequence, both a source and a destination sequence, or two source sequences. Sequential processing terminates when a length (element count) is exhausted. Other pointer operators, called *enter-edit operators*, serve to invoke one or more instances of another class of pointer operators, called edit-mode operators.

Typical pointer operators require initial stack arguments that specify the length, the source element sequence, and the destination element sequence. A source or destination descriptor defines the first element of the target sequence; such a descriptor is generally called a *pointer*. More formally, pointer (in the context of data descriptors) is defined as the type union IndexedCharDD or IndexedWordDD. A source element sequence can be contained in an operand or designated pointer; a destination sequence is always designated by a pointer.

Some operators (pack, input convert, string isolate) read a source and generate a result operand on the stack. These stack results are not defined to be destinations.

The char_index of an IndexedCharDD must not exceed five or eleven for EBCDIC or hex, respectively.

Element_size Conventions

The element size for the source and/or destination sequence can be specified by the element_size in the IndexedCharDDs, inferred by default, or fixed by the operator. Only the translate operator can manipulate source and destination sequences of different element sizes. The word-transfer operators always operate on single words, however, they can accept indexed descriptors of any valid element_size. The unpack operators require a source operand and handle it as hex.

All pointer operators, except for word-transfer, deal with either 4 or 8-bit characters. The following element_size adjustments are made (but the operand source of an unpack operator is always 4-bit):

- If there are both source and destination arguments (or two source arguments), and only one is an IndexedCharDD, the element_size of the IndexedCharDD is applied to the other argument.
- If there is no IndexedCharDD argument, then any source or destination is assumed to be EBCDIC.
- When a character element_size is applied to a source or destination word descriptor, that argument effectively becomes an IndexedCharDD; the index must not exceed 2**16-1.

For word-transfer operators, the element_size specified in the source and destination descriptors is unaltered and the operation handles single words. If an IndexedCharDD is used as source or destination and the char_index is nonzero, the descriptor is adjusted by setting char_index to zero and incrementing word_index; the resulting word index must not exceed 2**16-1.

For all pointer operators except word-transfer and translate, if the source and destination arguments are both IndexedCharDDs, the element_size values must be equal.

Segment Boundary

The term *segment boundary* refers to the edges of a virtual segment. A segment boundary is detected when a pointer operator attempts to access an array element outside the source or destination virtual segment. This detection causes the processor to generate an error or service interrupt.

Segment boundaries are determined from information in the ASDs (and in the page table for paged segments). A non-Data Word within the segment is reported separately via an error interrupt.

A segment boundary is not detected after the length has been exhausted, by compare-delete operators after a mismatch has been detected, or by scan or conditional transfer operators after a source character has failed to satisfy the condition.

Length Argument

When a pointer operator is invoked in initial mode, the comments in the subsequent paragraph apply. If a pointer operator is resumed in restart mode, the length argument must be updated by the interrupted operator. (In some restart situations, a length of zero is significant.)

The length argument must be an operand. It is integerized with rounding (RaI function) if required; the integer magnitude must not exceed MaxInx.

A negative length value is equivalent to zero. If length is less than or equal to zero, all pointer operators terminate without accessing any source element or transferring any destination element, no segment boundary is detected, and no service interrupts are generated. (The enter-single-edit operators do not terminate for zero-length input; rather, any edit operators that require a length do so.)

Source Argument

The source argument must be an operand or an IndexedDD of any valid element_size.

A source descriptor must have reindex enabled in all cases. All the elements accessed in the source sequence must be in operands.

All pointer operators reject an attempt to access a source element outside the source segment. For certain of these operators that unconditionally transfer data to a destination, the interrupt is a Source Boundary service request when the attempted access is to fetch the first element beyond the source virtual segment.

A source operand sequence begins with the high order character of the operand; it is interpreted according to element_size conventions defined above as EBCDIC sequence of 6 (or 12) characters or a hex sequence of 12 (or 24) characters, for a single (or double) precision operand, respectively. The operand is concatenated with itself as required to form a sequence of indefinite length.

Short Source Operators

The string-isolate, pack, and input-convert operators are special in that the source sequence is short (never more than 25 characters) and the result is left on the stack as an operand rather than be moved to a destination sequence. Some of these operators interpret one character or one zone field as a sign.

If a short-source operator is interrupted and the code sequence is subsequently resumed, the operator is repeated from the beginning.

Destination Argument

The destination argument must be an IndexedDD of any valid element_size; reindex must be enabled.

Any word modified in the destination sequence of a character transfer operator must contain an operand; any destination word written by a word transfer operator must contain a Data Word. Character transfer operators preserve the tags of destination words; word transfer operators copy the tag of the source word into the destination word.

An operator is said to *require a destination* if a destination stack argument is specified, except for the Enter Single Edit operators. These operators require a destination if the subsequent edit operator conditionally or unconditionally transfers data. All operators that require a destination require a write enabled destination pointer.

All pointer operators reject an attempt to access a destination element outside the destination sequence. If the attempt is to store into the first element beyond the destination segment, the interrupt is a Destination Boundary service request.

Source1 and Source2 Arguments

The compare operators process two sources, rather than a source and a destination. Source2 is a source defined previously. Source1 takes the place of a destination; it must be an IndexedDD (not an operand). Both sources require that reindex be enabled.

Overlapping Source and Destination

A source and destination are said to overlap if both arguments are IndexedDDs into the same segment and the displacement (index difference) between them is less than the effective length (number of elements transferred).

The effect of an overlapped unconditional word or character transfer depends upon the direction and magnitude of the displacement. In the following, D represents the displacement expressed as destination element index minus the corresponding source element index. L represents the transfer length.

```
      D <= -L:    No overlap
-L  <  D <=  0:   Destination sequence overwrites L+D source elements.
 0  <  D <   L:   Destination and source contents are undefined.
 L  <= D:        No overlap
```

Conditional (character-relational or set-membership) transfer operators are subject to the same constraints as unconditional transfers, but L in the previous specification is the number of elements actually transferred.

For translates involving 4-bit characters, the displacement d is in 4-bit characters from the initial source to the initial destination. For translation of L characters, the overlap cases and effects are as follows (where /2 indicates halving with truncation):

```
4- to 4-bit:     -L < d <= 0 :   L+d source elements overwritten
                  0 < d <  L :   Undefined

4- to 8-bit:    -2L < d <= 1-L:  min(L,2L+d) source elements overwritten
                1-L < d <  L  :  Undefined

8- to 4-bit:     -L < d <= 1 :   (L+d+1)/2 source elements overwritten (for L>0)
                  1 < d <  2L:   Undefined

8- to 8-bit:     -L < D <= 0 :   L+D source elements overwritten
                  0 < D <  L :   Undefined
```

For edit operators, the source and destination overlap if -Ld < D < Ls, where Ld and Ls are the destination and source length, respectively. (Note that for data-transferring edit operators, Ld >= Ls.) The result of overlapped editing is undefined if D > Ls-Ld. In a table-edit sequence, each group of consecutive edit operators (other than skips) is to be considered a unit for the application of this test. Overlap considerations do not apply directly to skip operators, however, skip operators in an edit table change the initial pointer displacements for subsequent operators; one operator's destination element might become another operator's source element.

Update

Many pointer operators occur in both *delete* and *update* variations. The delete forms consume all their stack arguments and do not leave updated results on the stack (although they may leave other results on the stack). The update forms leave on top of the stack an updated source and/or destination argument (if applicable) and the length (if relevant).

An updated length result is a single_integer indicating the number of elements remaining to be processed. It is produced by update operators that may terminate before the length is exhausted. If the initial length is negative, the updated length is zero.

An updated source operand is the original operand circularly rotated left so that the high ordered element is the one that would have been processed next.

A source or destination descriptor is updated as an indexed descriptor that references the next source or destination element that would have been processed. The updated descriptor reflects any adjustments made according to the element_size conventions previously defined. Word-transfer operators may adjust the char_index and word_index values; all other operators change any IndexedWordDD into an IndexedCharDD. These adjustments occur even when length less than or equal zero, but they are undefined for the special case of RSTF introduced by EXPU or EXSU.

An updated descriptor from a forward-acting operator can designate the first element beyond the segment.

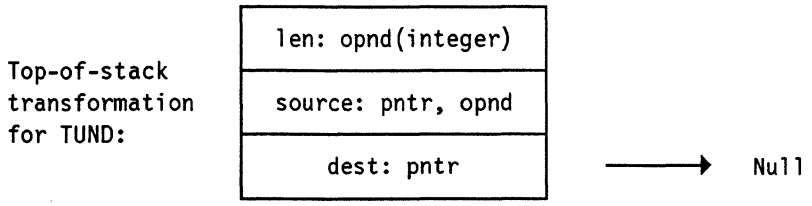
The field-width limits in a descriptor are 2**16-1 for the word_index field in an IndexCharDD and 2**20-1 for the index field in an IndexedWordDD. If the word index value to be updated into a descriptor exceeds the limit, an error interrupt is generated. The A16 generates the interrupt at any point in the sequence processing where the word index would exceed the limit.

Most pointer operators can be interrupted (for example, at a page boundary) in such a way that the operation can be resumed where it stopped. Both update and delete operators are subject to such interruption; the stack arguments are configured for resuming the operator in initial (for short-source operators) or restart (for all other pointer operators) mode. In restart mode, the original arguments are left on the stack with a progress count included in the restart information.

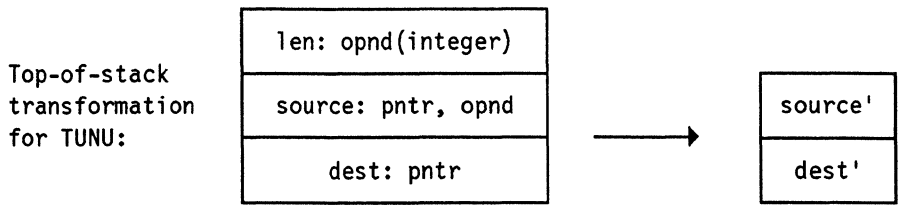
Unconditional Character-Transfer Operators

Unconditional character transfer operators transfer hex or EBCDIC characters from the source to the destination. The number of characters transferred is specified by the length. TFFF is left in an undefined state.

TUND <transfer characters unconditional delete>



TUNU <transfer characters unconditional update>



Character Relational Operators

Character relational operators sequentially apply a relational comparison of each source character to a delimiter character supplied by a stack argument until the length is exhausted or a relation fails. TFFF indicates the cause of termination: it is reset to 0 if a relation fails and set to 1 if the length is exhausted (all source characters satisfy the relation or the initial length is less than or equal to zero).

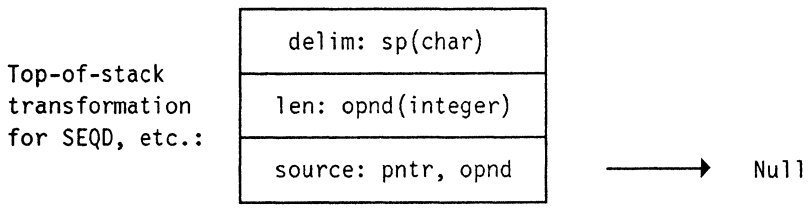
The delimiter argument must be a single precision operand; it is interpreted as a single right-justified character (EBCDIC or hex according to the effective element size of the source); all bits in the delimiter word except those in the delimiter character are ignored.

The binary value of each source character is compared to the binary value of the delimiter character. The operator names specify the relation of the source character to delimiter that must hold for the operation to continue. For example, the SLSU operator scans across source characters less than the delimiter.

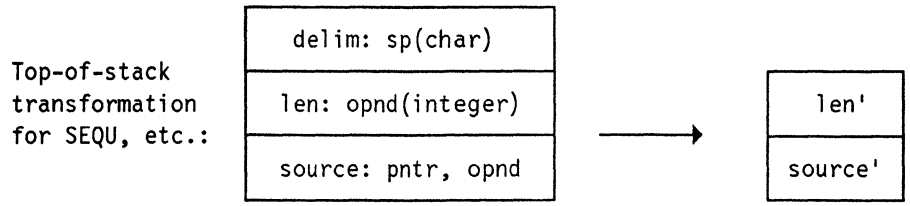
Scan Operators

Character relational scan operators sequentially compare each source character to the delimiter character as defined previously.

SEQD (scan while equal delete)
SNED (scan while not equal delete)
SGTD (scan while greater delete)
SGED (scan while greater or equal delete)
SLED (scan while less or equal delete)
SLSD (scan while less delete)



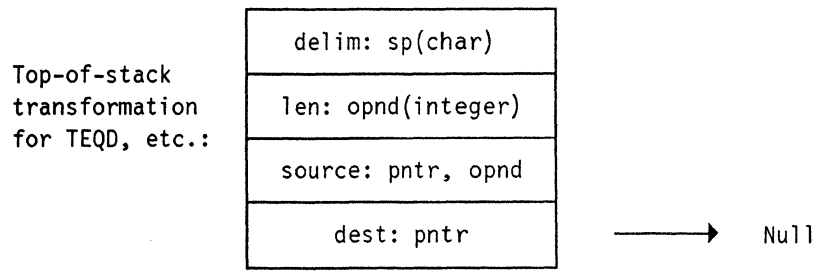
SEQU (scan while equal update)
SNEU (scan while not equal update)
SGTU (scan while greater update)
SGEU (scan while greater or equal update)
SLEU (scan while less or equal update)
SLSU (scan while less update)



Transfer Operators

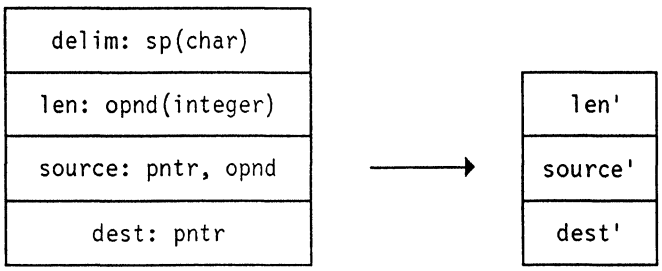
Character relational transfer operators sequentially compare each source character to the delimiter character as defined previously. Each source character that satisfies the relation is transferred to the destination sequence.

TEQD (transfer while equal delete)
TNED (transfer while not equal delete)
TLED (transfer while less or equal delete)
TLSD (transfer while less delete)
TGTD (transfer while greater delete)
TGED (transfer while greater or equal delete)



TEQU (transfer while equal update)
TNEU (transfer while not equal update)
TGTU (transfer while greater update)
TGEU (transfer while greater or equal update)
TLEU (transfer while less or equal update)
TLSU (transfer while less update)

Top-of-stack transformation for TEQU, etc.:



Character-Sequence Compare Operators

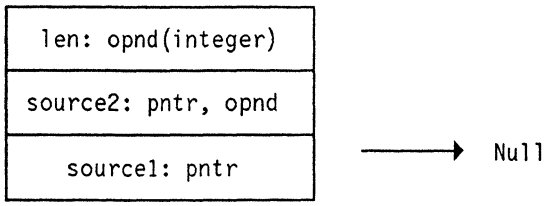
Character sequence compare operators apply a relational comparison of the source1 sequence to the source2 sequence. TFFF is set to 1 if the relation is satisfied and reset to 0 if the relation fails. The operator name specifies the relation being tested.

The binary values of each corresponding source1 and source2 character are compared. The two sequences are equal if, and only if, each source1 character is equal to the corresponding source2 character for the specified length or the initial length is zero. Source1 is strictly less (greater) than source2 if, and only if, for the first (left-most) pair of unequal characters, the source1 character is strictly less (greater) than the source2 character.

The following operators terminate when the actual relation is determined.

CEQD (compare characters equal delete)
CNED (compare characters not equal delete)
CGTD (compare characters greater delete)
CGED (compare characters greater or equal delete)
CLED (compare characters less or equal delete)
CLSD (compare characters less delete)

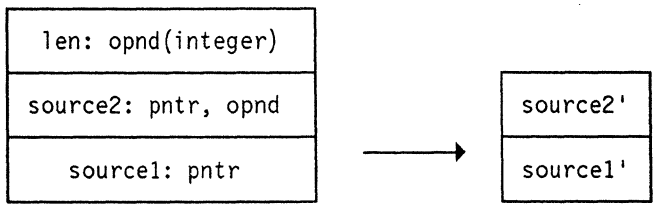
Top-of-stack transformation for CEQD, etc.:



The following operators terminate only when the length is exhausted. They are all of limited utility and are obsolete. Each updated IndexedCharDD references the first character after the end of the sequence, as determined by the length. If the operator is interrupted and resumed after the relation (TFFF state) has been determined, restart mode is used in such a way that TFFF is not modified. MCP/AS (Extended) A Series systems are not required to access and tag check the source words following the point at which the relation is determined, however, the A16 does perform these checks.

CEQU (compare characters equal update)
CNEU (compare characters not equal update)
CGTU (compare characters greater update)
CGEU (compare characters greater or equal update)
CLEU (compare characters less or equal update)
CLSU (compare characters less update)

Top-of-stack transformation for CEQU, etc.:



Character Set Membership Operators

Character set membership operators test source characters for membership in a character set supplied by a stack argument. The relations applied are inclusion and exclusion. Source characters are sequentially tested until the relation fails or the length is exhausted. TFFF indicates the cause of termination: it is reset to 0 if a relation fails and set to 1 if the length is exhausted (all source characters satisfy the membership criterion or the initial length is less than or equal to zero).

The character set argument must be an IndexedSingleDD, which designates the first word of the character set. Reindex must be enabled in this descriptor. Every word accessed in the set must be an operand. Every word accessed in the set must lie within the designated virtual segment. Note that in pre-MCP/AS (Extended) A Series systems, every word accessed in the set had to be within the actual segment containing the first word, meaning the character set words could not be split by a page boundary for those systems.

The character set is interpreted as a bit vector indexed by the source character. If the selected bit is 1, the character is included in the set; otherwise, it is excluded from the set. The bit is located by the following expression, where Set represents the character set argument and SetSeg the segment designated by that argument:

```
SetSeg[Set.index+WordIndex(c)].[BitIndex(c):1]
```

WordIndex and BitIndex are computed from the binary representation of the source character (c) as follows:

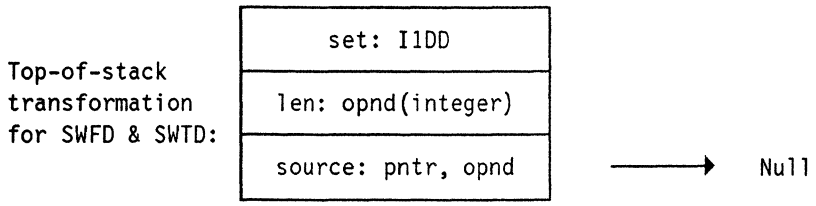
```
EBCDIC      WordIndex = value of three high-order bits
            BitIndex  = 31 - (value of five low-order bits)

hex         WordIndex = 0
            BitIndex  = 31 - (4-bit value)
```

In the following operator names, the relation "while source included in set" is called "while true" and "while excluded from set" is called "while false".

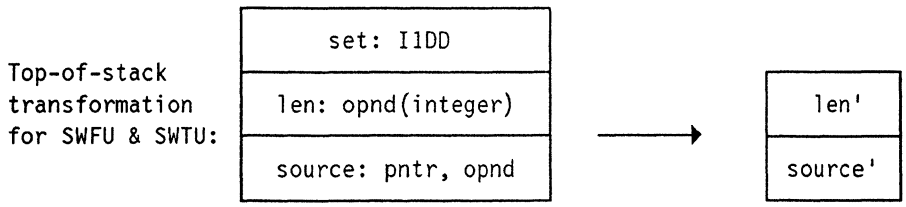
Scan Operators

Character set membership scan operators apply the sequential membership test of each source character to the character set as defined previously.

SWFD <scan while false delete>
SWTD <scan while true delete>



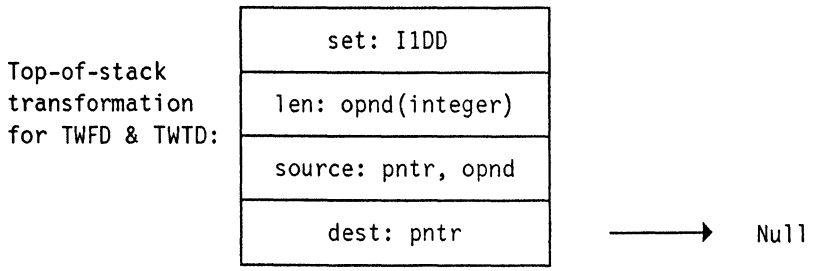
SWFU <scan while false update>
SWTU <scan while true update>



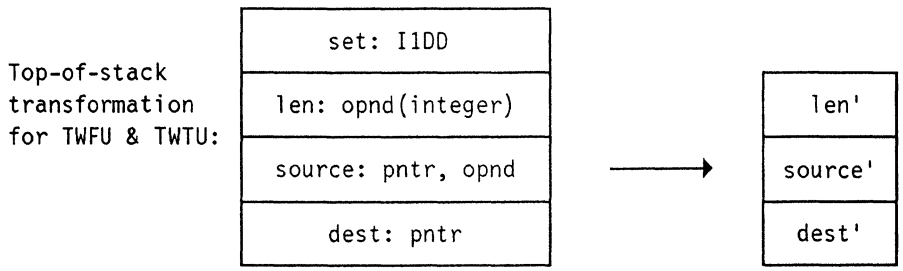
Transfer Operators

Character set membership transfer operators apply the sequential membership test of each source character to the character set as defined previously. Each source character that satisfies the membership criterion is transferred to the destination sequence.

TWFD <transfer while false delete>
TWTD <transfer while true delete>



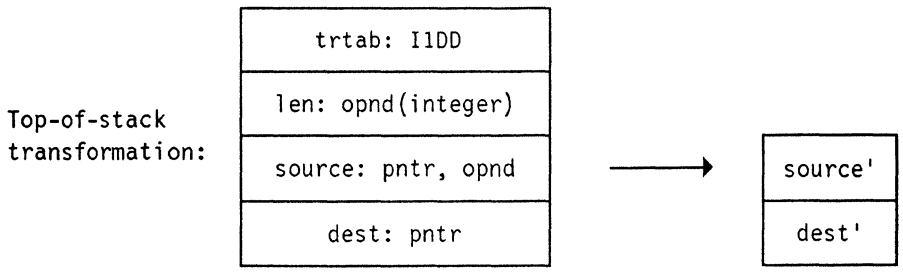
TWFU <transfer while false update>
TWTU <transfer while true update>



Character Translate Operator

TRNS sequentially accesses characters from the source sequence, maps each character into a specified character set, and stores the translated character into the destination sequence. The character set mapping is indicated by a translate table argument.

TRNS <translate>



The translate table must be an IndexedSingleDD, which designates the first word of the translate table. Reindex must be enabled in this descriptor. Every word accessed on the table must be an operand. Every word accessed in the table must lie within the designated virtual segment. Note that in pre-MCP/AS (Extended) A Series systems, every word accessed in the table had to be within the actual segment containing the first word, meaning that the translate table words could not be split by a page boundary for those systems.

The translate table is interpreted as an array of words, each containing four right-justified 8-bit characters; it is indexed by the source character. The selected 8-bit character is stored into an EBCDIC destination or the four low-order bits of the character are stored into a hex destination. The character is located by the following expression, where Table represents the translate table argument and TableSeg represents the segment designated by that argument.

```
TableSeg[Table.index+WordIndex(c)].[FieldIndex(c):8]
```

WordIndex and FieldIndex are computed from the binary representation of the source character (c) as follows:

```
EBCDIC      WordIndex  =  value of six high-order bits
            FieldIndex =  31 - 8*(value of two low-order bits)

hex         WordIndex  =  value of two high-order bits
            FieldIndex =  31 - 8*(value of two low order bits)
```

TRNS leaves the updated source on top of the stack and the updated destination second from top of the stack.

Character Sequence Extraction Operator

SISO extracts a character sequence from the source, creates an operand containing the extracted sequence right-justified with leading zero-fill if required, and leaves the operand on top of the stack. The length specifies the number of characters in the extracted sequence.

Top-of-stack transformation:

The result is a single or double precision-operand depending on the length and the source character type. If the source is EBCDIC, the result is single for length less than or equal to six and double for {7 to 12}. If the source is hex, the result is single for length less than or equal to twelve and double for {13 to 24}. The length must not exceed 12 or 24 for EBCDIC and hex source, respectively.

Decimal Character Sequence Operators

Decimal character sequence operators interpret hex or EBCDIC sequences as digit sequences and provide conversion functions among various numeric representations.

For definitions of digit sequences and decimal sequence operands, see "Decimal Sequence Operands" in Section 2.

The hex representation of a digit sequence is as a hex sequence of the corresponding digit values. The EBCDIC representation of a digit sequence is an EBCDIC sequence in which the numeric field (low-order four bits) of each character contains a digit value. The high-order four bits are called the zone field; zone fields are significant in some operators, but they do not form part of the digit sequence.

A signed decimal integer is represented as a digit sequence and a sign value. Hex "D" represents a negative sign; any other 4-bit value represents a positive sign. The sign may be placed at either the left (high-order) or right (low-order) end of the sequence. A signed sequence of n digits is represented in hex as a sequence of n+1 characters; the sign is the leftmost or rightmost character. A signed n-digit sequence is represented in EBCDIC as a sequence of n characters; the sign occupies the zone field of the leftmost or rightmost character. (Operand digit sequences are always unsigned; EXTF can be used to hold the sign.)

There are three groups of decimal character sequence pointer operators. The pack operators transform a hex or EBCDIC source sequence into a right-justified operand digit sequence. The unpack operators transform a left-justified operand digit sequence into a hex or EBCDIC destination sequence. The input-convert operators are similar to pack, but the integer value of the source sequence is transformed to binary representation.

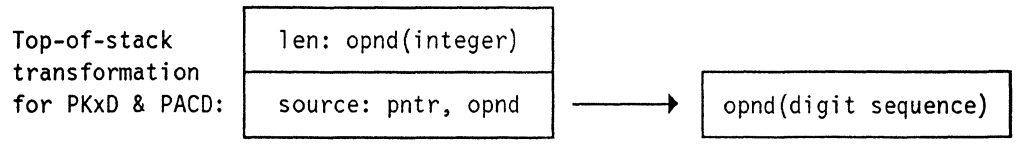
Two pairs of operators, PACD/PACU and ICVD/ICVU, handle a hex source sequence as either unsigned or left-signed, depending upon the value of the first character. A nondigit value is considered as a sign; a digit value is a digit. The sign is not counted in the length.

Pack Operators

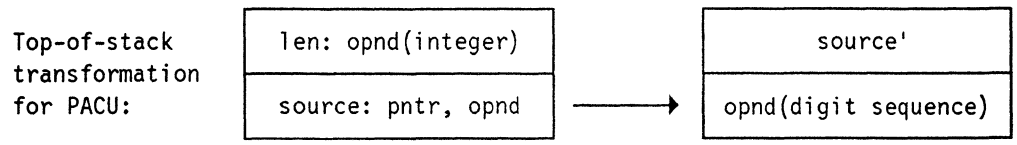
Pack operators perform a conversion from the source EBCDIC or hex digit sequence to a decimal operand containing the corresponding digit sequence right-justified with leading zero-fill. The operand is left as a result on the stack. Nondigits in a hex source sequence (other than a sign character) or in the numeric field of an EBCDIC source sequence are transferred unmodified to the operand digit sequence.

The result is a single precision operand if length less than or equal to twelve and double precision for {13 to 24}. The length must not exceed 24.

PKUD <pack unsigned delete>
PKLD <pack left-signed delete>
PKRD <pack right-signed delete>
PACD <pack delete>



PACU <pack update>



PACD is obsolescent, superseded by PKUD, PKLD, and PKRD. PACU is obsolete, except as part of an obsolescent code sequence that is superseded by PKRD.

PKUD leaves EXTF and TFFF in undefined states. All other pack operators set both EXTF and TFFF: true = negative and false = positive or unsigned. If the length argument is less than or equal zero, EXTF and TFFF are set false.

If the length is greater than zero, the source is hex, and there is a sign, the number of characters read from the hex sequence is one greater than the length value. A sign is always present for PKLD and PKRD and never present for PKUD. A sign is present for PACx when the leftmost hex character is a nondigit. If the length is less than or equal to zero, no source characters are read and the digit-sequence result is zero.

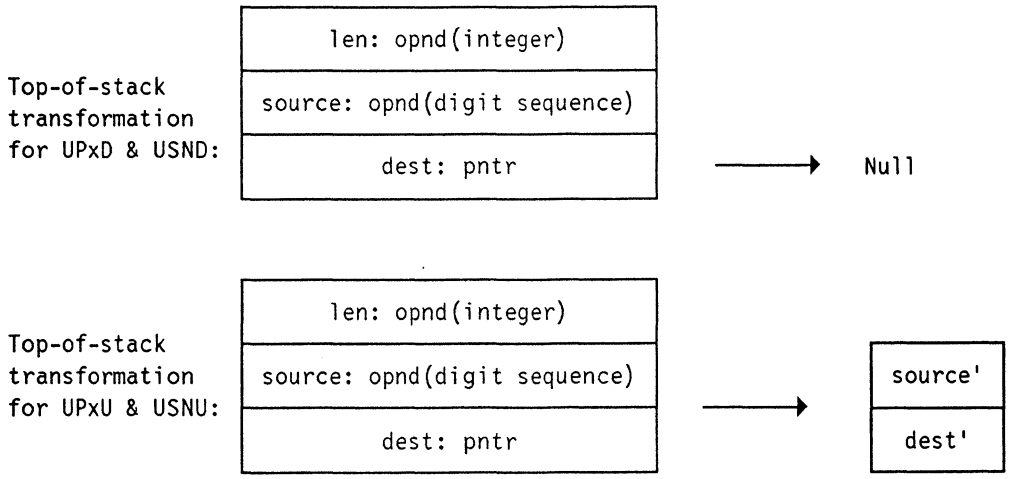
The following are the sign locations for these operators:

PKUD:	none
PKLD, EBCDIC:	zone of leftmost character
PKLD, hex:	leftmost character
PKRD, EBCDIC:	zone of rightmost character
PKRD, hex:	rightmost character
PACx, EBCDIC:	zone of rightmost character
PACx, hex:	leftmost character if nondigit, else none

Unpack Operators

Unpack operators interpret the source operand as a left-justified digit sequence and

store the corresponding hex or EBCDIC digit sequence into the destination. Nondigits in the operand digit sequence are transferred unmodified to the hex characters or the numeric field of the EBCDIC characters in the destination sequence.



The element_size convention for unpack is that the source operand is unconditionally handled as hex. If the destination is an IndexedWordDD, it is changed to an EBCDIC IndexedCharDD. The source must be an operand. The length must not exceed 24.

Unpack-Unsigned Operators

Unpack-unsigned operators store the destination digit sequence without a sign. For an EBCDIC destination, the zone field of each character is set to hex "F". For a hex destination, the digit sequence is stored with no sign character.

UPUD <unpack unsigned delete>
UPUU <unpack unsigned update>

Unpack-Signed Operators

Unpack signed operators store the destination digit sequence with a sign; the sign is determined by EXTF (external sign flip-flop), where true equals negative and false equals positive. The operator mnemonics and names are listed as follows with the location of the sign for hex and EBCDIC sequences:

	Hex	EBCDIC
UPLD (unpack left-signed delete)	left	left zone
UPLU (unpack left-signed update)	left	left zone
UPRD (unpack right-signed delete)	right	right zone
UPRU (unpack right-signed update)	right	right zone
USND (unpack signed delete)	left	right zone
USNU (unpack signed update)	left	right zone

Hex "C" and Hex "D" are used as the positive and negative sign characters, respectively. For an EBCDIC destination, the sign is inserted into the zone field of the rightmost or leftmost character, depending on the operator, and all other zone fields are set to hex "F". For a hex destination and length greater than zero, the sign is inserted as the leftmost or rightmost character, depending on the operator; length+1 hex characters are transmitted to the destination sequence. If the length is less than or equal to zero, no characters are transmitted to the destination sequence.

Unpack operators can be considered *short destination* operators, analogous to short-source operators. If interrupted at a page boundary, the A16 repeats them from the beginning after both pages are present. (Although restart mode is used for convenience, the progress count is set to zero to force repetition from the beginning.)

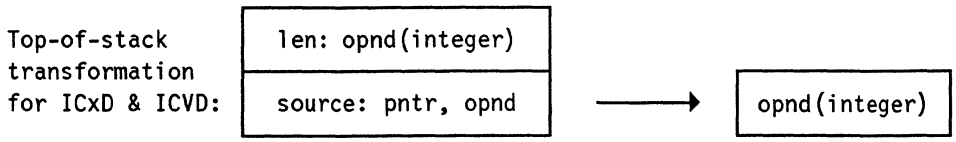
USND and USNU are obsolescent, superseded by UPLD/UPRD and UPLU/UPRU.

Input Convert Operators

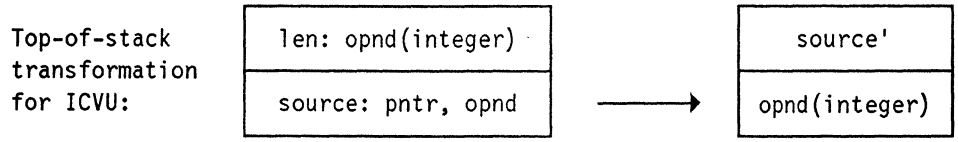
Input convert operators perform a conversion from the source EBCDIC or hex digit sequence to a numeric operand containing the signed integer value of the corresponding digit sequence. The operand is left as a result on the stack. TFFF and EXTF are left in undefined states.

The length must not exceed 23. If the length is less than 12, a single_integer is produced. If the integer absolute value of the source exceeds 8**13-1, a double_integer is produced. If the length exceeds 11, but the result absolute value is less than 8**13, the A16 represents the value as an extended single_integer; however, some other A Series systems represent the value as a single_integer or a double_integer.

ICUD <input convert unsigned delete>
ICLD <input convert left-signed delete>
ICRD <input convert right-signed delete>
ICVD <input convert delete>



ICVU <input convert update>



ICVD is obsolescent, superseded by ICUD, ICLD, and ICRD. ICVU is obsolete except as part of an obsolescent code sequence that is superseded by ICRD.

If the length is greater than zero, the source is hex, and there is a sign, the number of characters read from a hex sequence is one greater than the length value. A sign is always present for ICLD and ICRD and never present for ICUD; a sign is present for ICVx when the leftmost hex character is a nondigit. If the length is less than or equal to zero, no source characters are read and the binary integer result is positive zero.

The sign locations for these operators are the same as for the corresponding pack operators:

ICUD:	none
ICLD, EBCDIC:	zone of leftmost character
ICLD, hex:	leftmost character
ICRD, EBCDIC:	zone of rightmost character
ICRD, hex:	rightmost character
ICVx, EBCDIC:	zone of rightmost character
ICVx, hex:	leftmost character if nondigit, else none

If the source digit sequence contains a nondigit, the result value is undefined, except that a sequence of all hex "F" characters is equivalent to a sequence of nines. (Note that "digit sequence" does not include the zone fields of an EBCDIC source or a sign character in a hex source.)

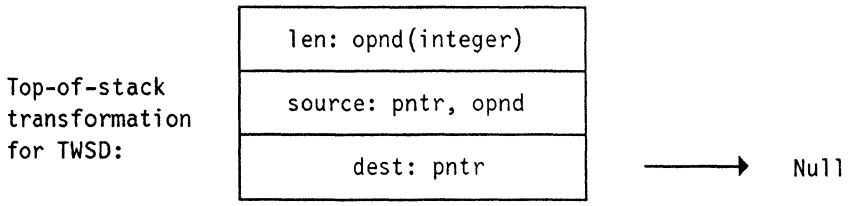
Word Transfer Operators

Word transfer operators transfer word elements from the source to the destination. The number of words is specified by the length. The tag of each source word is copied to the corresponding destination word.

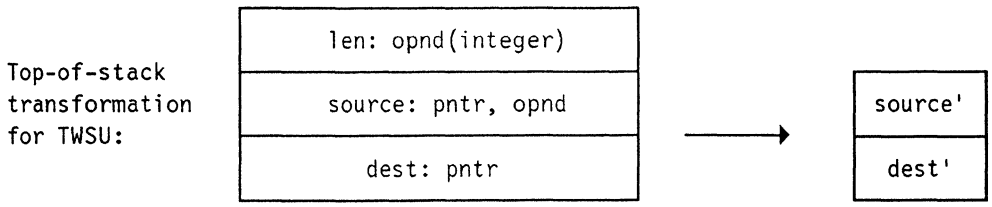
A source operand is interpreted as a word or pair of words concatenated with itself indefinitely.

Source and destination IndexedCharDDs, if not already word-aligned, are advanced to the next word boundary.

TWSD <transfer words delete>



TWSU <transfer words update>



Edit Operators

Edit mode operators can be considered sub-operators invoked by a special class of pointer operators, the *enter-edit* operators. Most edit operators process source and/or destination characters sequentially until a length is exhausted.

Enter-Edit Operators

There are two modes in which edit operators are executed; each is initiated by an enter edit operator. The enter edit operator provides the destination and (except for EXPU) source. It may specify update, which causes an updated source (if any) and destination argument to be left on top of the stack at termination of edit mode.

1. Table edit mode:

   A sequence of edit operators is executed until terminated by ENDE (end edit). Each acts on the source and destination supplied by the table enter edit operator; length is a parameter for each edit operator requiring it. If update is specified, the updated source and destination are left on the stack by ENDE.

   Each edit operator that uses the source/destination updates it internally at termination so that a group of edit operators may sequentially process source/destination characters. Character-skip operators may advance or back up the source/destination to alter the normal sequential processing.

2. Single edit mode:

   A single edit operator acts on the source and destination supplied by the enter single edit operator. Length is also supplied as a stack argument at entry, whether or not it is required by the edit operator. If update is specified, the updated source and destination are left on the stack at termination of the edit operator.

The enter-edit operators validate argument types and perform any needed element_size adjustment, as defined previously for other pointer operators.

With the exception of EXPU, all enter-edit operators set FLTF = 0 when executed in initial mode. When restarted, these operators preserve the FLTF value.

Enter-Table-Edit Operators

Enter table edit operators supply the source and destination sequences and a reference to the sequence, or table, of edit operators to be executed. Each edit operator acts on the source and/or destination supplied at entry; length is a parameter for each edit operator requiring it.

The edit table must be an unpaged IndexedDD, which is interpreted as usual, except that the element_size field is ignored and the index field is subdivided as follows:

```
          [39: 1]   Must be zero
esi       [38: 3]   Edit table syllable index of the first edit operator;
                    must be in {0 to 5}
          [35: 3]   Must be zero
ewi       [32:13]   Edit table word index of the word containing the first
                    edit operator
```

(The set of valid edit-table descriptors includes IndexedSingleDDs and EBCDIC IndexedCharDDs that have index or word_index values less than 2**13.)

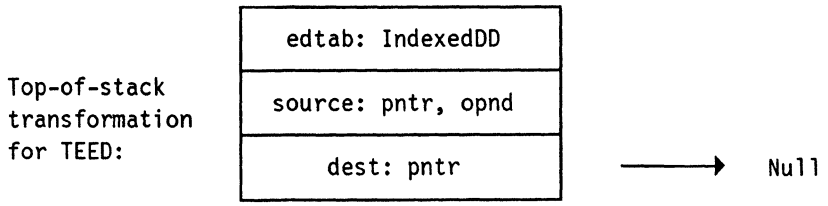
The edit table descriptor must be reindex enabled.

Edit operators (and their parameters) are fetched from the edit table, starting from the esi syllable of the ewi word, until completion of an ENDE (end edit) operator. The normal code stream is then resumed with the operator following the enter table edit operator.

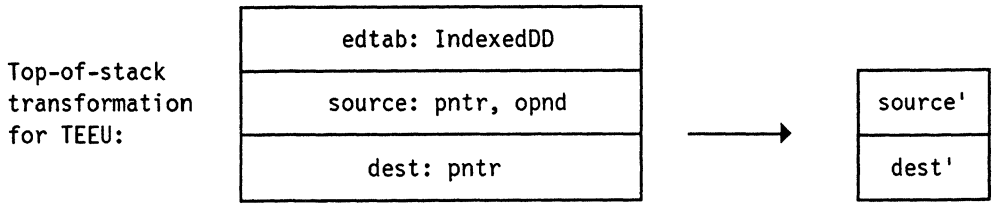
Each edit table word that is fetched must have a tag = 0. The table cannot extend beyond word index 2**13-1.

When a table edit sequence is interrupted and resumed, the A16 uses restart mode with an additional stack argument to provide a progress count and to reestablish the FLTF value. The updated table argument points to the edit operator that generated the interrupt.

TEED <table enter edit delete>



TEEU <table enter edit update>



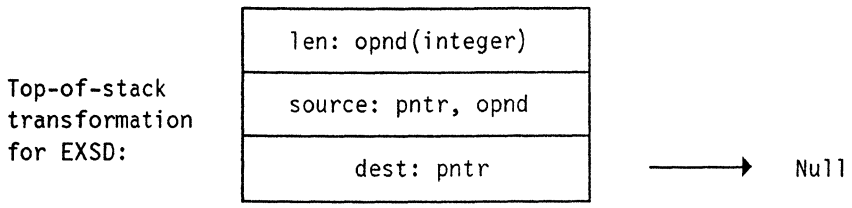
Enter-Single-Edit Operators

Enter single edit operators supply the destination sequence, occasionally the source sequence, and the length for the edit operator that follows it in the code stream. Each argument must be on the stack and must meet type restrictions, although it may not be required by the edit operator.

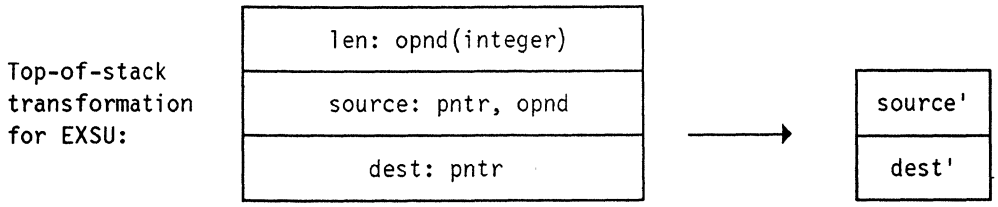
All edit operators requiring length terminate without transferring any characters if the length is zero.

The enter-single-edit operators cannot introduce the edit operators that insert a character conditionally based on FLTF; the excluded operators are INSC and ENDF. The EXSD and EXSU operators can introduce the move operators that sense and set FLTF (MFLT and MINS), although FLTF is unconditionally reset at the beginning of the operator and discarded at the end. When an edit operator introduced by EXSx is interrupted and resumed, the A16 uses restart mode with an additional stack argument to reestablish the FLTF value and to provide a progress count.

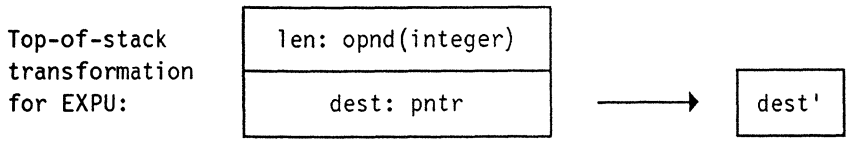
EXSD <execute single edit operator delete>



EXSU <execute single edit operator update>



EXPU <execute single edit operator, single pointer update>



No source is provided for EXPU; the subsequent edit operator must not require a source. (The excluded edit operators are MCHR, MVNU, MINS, MFLT, SFSC, and SRSC.)

The element_size convention applied for EXPU is that an IndexedWordDD destination pointer is changed to an EBCDIC IndexedCharDD.

No delete form of single pointer enter edit operator is provided.

Edit Mode Operators

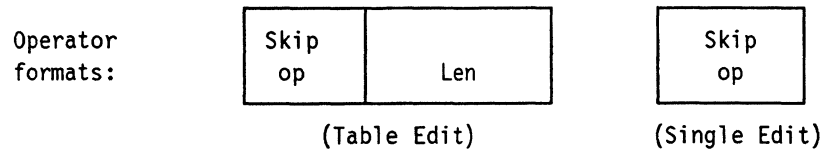
The following paragraphs define the operators that are executed in edit mode (under the control of an enter-edit operator).

Each edit operator is permitted to update its arguments and enforce all the checking associated with pointer update. In particular, any error due to an unrepresentable updated pointer can be trapped, even if the resulting pointer would have been deleted without further use.

Character Skip Operators

Character skip operators advance or reverse the source or destination sequence. Length indicates the number of characters to be skipped.

Length is a parameter for table edit mode only:



A skip operation on a pointer alters the pointer so that it designates a different element in the same sequence; the length value designates the extent of the change and the operator (forward or reverse) its direction. The A16 implements the operation as a reindexing of the pointer.

The skip operators use aAPIX and are subject to the same constraint as INDX. The operations must abort when the resulting virtual index cannot be represented in an indexed DD; the A16 also causes them to abort if the resulting pointer designates an element beyond last+1.

Skip Forward

SFSC <skip forward source characters>
SFDC <skip forward destination characters>

Character skip forward operators advance the source or destination sequence. A source operand is circularly rotated left by length characters. An IndexedCharDD is incremented by length characters.

The maximum length value that can be applied to a pointer is one that causes the pointer to designate the first element beyond the virtual segment. However, software cannot count on an error exception for all larger length values.

The SFSC operator cannot be introduced by EXPU.

Skip Reverse

SRSC <skip reverse source characters>
SRDC <skip reverse destination characters>

Character skip reverse operators back up the source or destination sequence. A source operand is circularly rotated right by length characters. An IndexedCharDD is decremented by length characters.

The maximum length value that can be applied to a pointer is one that causes the pointer to designate the first element of the virtual segment.

The SRSC operator cannot be introduced by EXPU.

Character Insert Operators

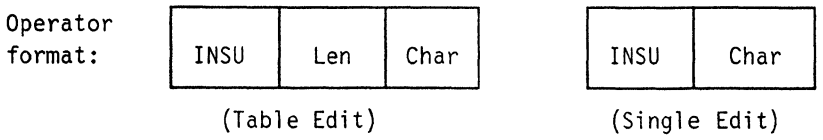
Character insert operators store a character or a sequence of characters into the destination sequence, in some cases conditionally based on the value of FLTF (float flip-flop) and EXTF (external sign flip-flop). Each character is a parameter, except for a fixed sign character.

Several insert operators do not allow a hex destination. Those that do store only the numeric field of a parameter character.

INSU <insert unconditional>

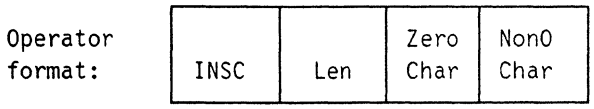
INSU stores a sequence composed of length repetitions of the parameter character (Char) into the destination.

Length is a parameter for table edit mode only:



INSC <insert conditional>

INSC stores a sequence composed of length repetitions of a selected parameter character into the destination. If FLTF equals zero, ZeroChar is selected; if FLTF equals one, NonZeroChar is selected.



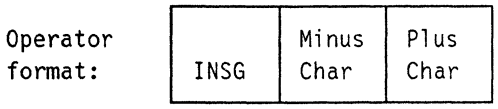
The INSC operator is defined only for table edit.

INOP <insert overpunch>

INOP stores hex"D" into the zone field of the destination character if EXTF=1; the destination character is not altered if EXTF equals zero. Note that in either case the destination IndexedCharDD is advanced one character. The destination element_size must not be hex.

INSG <insert display sign>

INSG stores MinusChar into the destination if EXTF equals one and stores PlusChar if EXTF equals zero. The destination element_size must not be hex. MinusChar and Pluschar are parameters:

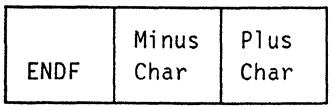


ENDF <end float>

If FLTF equals zero, ENDF stores a selected parameter character into the destination; MinusChar is selected if EXTF equals one, and PlusChar is selected if EXTF equals zero. If FLTF equals one, no character is stored and the destination IndexedCharDD is not advanced. FLTF is unconditionally reset to zero.

MinusChar and PlusChar are parameters:

Operator format:



The ENDF operator is defined only for table edit.

Character Move Operators

Character Move operators transfer characters from source to destination with editing. Some move operators conditionally store into the destination a sequence of repeated parameter characters based on the value of FLTF (float flip-flop), the source character, and EXTF (external sign flip-flop).

Move operators cannot be introduced by EXPU.

If the destination element_size is hex, only the numeric field of a parameter character is stored.

MCHR <move characters>

MCHR transfers length characters from the source to the destination. Length is a parameter for table edit mode only:

Operator format:

(Table Edit) (Single Edit)

MVNU <move numeric>

For an EBCDIC source and destination, MVNU transfers length numeric fields from the source to the destination, setting each zone field to hex"F". For a hex source and destination, MVNU transfers length hex characters (in this case MVNU is identical to MCHR).

Length is a parameter for table edit mode only:

Operator format:

(Table Edit) (Single Edit)

MINS <move with insert>

MINS performs a leading zero suppression function from the source to the destination for length source characters. In the following definition, the *source numeric field* is the numeric field of an EBCDIC character or the entire hex character.

Length is a parameter for table edit mode only:

Operator format:

(Table Edit) (Single Edit)

While FLTF equals zero and the value of the source numeric field is zero, the

ZeroChar parameter is transferred to the destination. If the value of the source numeric field is nonzero, FLTF is set to one, and the source numeric field is transferred as described in the next paragraph.

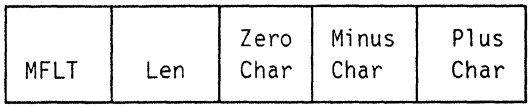
When FLTF equals one, the source numeric field is transferred to the destination and the zone field of an EBCDIC destination character is set to hex"F".

MFLT <move with float>

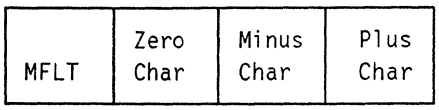
MFLT performs a signed leading zero suppression function from the source to the destination for length source characters. MFLT is functionally equivalent to MINS (move with insert) except for conditional insertion of a sign character into the destination sequence.

Length is a parameter for table edit mode only:

Operator format (Table Edit):



Operator format (Single Edit):



While FLTF equals zero and the value of the source numeric field is zero, the ZeroChar parameter is transferred to the destination.

If FLTF equals zero and the value of the source numeric field is nonzero, the PlusChar (if EXTF equals zero) or the MinusChar (if EXTF equals one) is inserted in the destination sequence, FLTF is set to one, and the source numeric field is transferred as defined for MINS.

When FLTF equals one, the source numeric field is transferred to the destination, as in MINS.

Note that the number of characters stored into the destination sequence may be length+1. Length characters are stored only if FLTF is initially zero and (for length characters) all source numeric fields are zero, or if FLTF is initially one.

Miscellaneous Edit Operators

RSTF <reset float flip-flop>

RSTF unconditionally resets FLTF (float flip-flop) to zero.

ENDE <end edit>

ENDE terminates table edit mode. The top-of-stack transformation for the TEED or TEEU operation is completed.

ENDE is defined only for table edit.

Pointer Index Restrictions

The following information summarizes restrictions that have already been stated or that follow from the stated restrictions.

The 16-bit word_index field in IndexedCharDDs restricts their use to the first 65535 (2**16-1) words of an actual segment. Operators that generate updated pointers cannot update the word index to 65536. For example, TRNS or TUNU cannot access the last character of the word at 65535; TWSU cannot access that word at all when the pointer is an IndexedCharDD. The delete form of non-edit operators can access all of that word, because update does not occur. Software must assume that edit operators have the same restrictions as other character update operators, because they may unconditionally update.

A sequence accessed with an IndexedCharDD cannot extend into or beyond the word with index 65536 (2**16).

With an IndexedWordDD, TWSD and TWSU can access through the word with index equals 2**20-1 or 2**20-2, respectively.

Vector Operators

A number of operators are provided to perform calculations that are typical of numerical (scientific and engineering) computation.

Vectors and Scalars

Each vector operator performs an iterative (or sometimes, in principle, parallel) computation on a set of elements of one or two virtual segments. Such a set of elements is called a vector; it is defined by an IndexedWordDD and a set of index values that are applied to it. In most cases the index values are an arithmetic sequence

```
j, j + k, j + 2*k, ...
```

in which j is the initial index value (supplied via the IndexedWordDD) and k is an integer constant called the *stride*. (Two operators allow the set of index values to be provided explicitly in a third vector.)

Some of the vector computations involve numeric values (operands) that are not in vectors; these are called scalars.

Vector Arguments

The vectors are named A, B, and (for index sets) I. When no index set is involved, vector A is specified by a pair of arguments called vector_A and stride_A. Vector_A is always an IndexedWordDD and stride_A is an integer. Vector_B is similarly specified by vector_B and stride_B, but in some cases vector_B can be an operand (in which case stride_B is not significant but must satisfy type and range restrictions). In two operators, a vector_I argument occurs instead of stride_A or stride_B; vector_I has an implicit stride of +1.

Any IndexedWordDD vector argument must have reindex enabled. The vector_A argument must be write-enabled for those operators that store into vector A.

The notation "B may be scalar" or "B must be vector" is used as shorthand for "vector_B must be an IndexedWordDD or operand" or "vector_B must be an IndexedWordDD", respectively.

A[x] denotes the element designated when the initial vector_A argument is reindexed by x. Lowercase i denotes an iteration index; A(i) denotes the ith element of vector A, counting from 1: A(i) = A[(i-1)*stride_A] normally; and A(i) = A[I(i)] denotes that an index set is used. The same notation is used for B, but when vector_B is an operand, B(i) = vector_B for all i.

Vector Length

All vector operators accept an argument named length, which specifies the vector length (iteration count). The length argument must be a single_integer; it must not exceed MaxLen in magnitude. If length is zero or negative, no iterations are performed; operators that return results return the corresponding input arguments.

Strides

Stride arguments must be single_integers that may be positive or negative but must not exceed MaxLen in magnitude. Strides are repeatedly applied as index values to the IndexedWordDDs, using common action aAPIX. Note that if stride is +1 and the vector argument is an IndexedDoubleDD, successive vector elements differ by two in the word index.

The initial vector descriptor index, the stride, and the length must be such that every access to a vector element falls within the designated virtual segment; an error is reported only if an out-of-bounds access is attempted. Furthermore, the product of stride*(length-1) must not exceed MaxLen for stride_A or stride_B, even when vector_B is an operand.

Vector Access

Vector elements are fetched and stored using the common actions aFOP and aSOP, respectively, with the current vector descriptor. The common action aAPIX is used to apply the stride or index-set value to each vector descriptor at each iteration of the operator.

Vector Arithmetic

The arithmetic performed by vector operators is as defined for the operators ADD, SUBT, MULT, MULX, and DIVD. Arithmetic exceptions occur as defined for those operators. (No vector operators generate Integer Overflow exceptions.)

Most vector operators can perform single and double precision computations, depending on the types of the arguments. Normal precision rules are applied to individual arithmetic operations. The result of addition, subtraction, multiplication, or division is double precision if either or both inputs are double; the result is single if both inputs are single. However, those operators that perform multiple arithmetic operations (multiply and add) in the same iteration are so specified that both operations are in the same precision.

The term width is defined as the number of words occupied by each vector element or scalar. The terms wA and wB denote widths of A and B. It is always required that the values to be accumulated or stored be of the same width as the accumulator or vector; no implicit type coercion is performed.

Vector Assignment Operators

These operators generate an output vector A by computing each element independently; the only vector elements that contribute to the new value of A(i) are A(i) or B(i) or both.

Vector_A must be write enabled.

Most operator definitions use succinct notation, in which operations on the vector name stand for operations on all the individual elements.

For example:

```
A := A * B
```

means

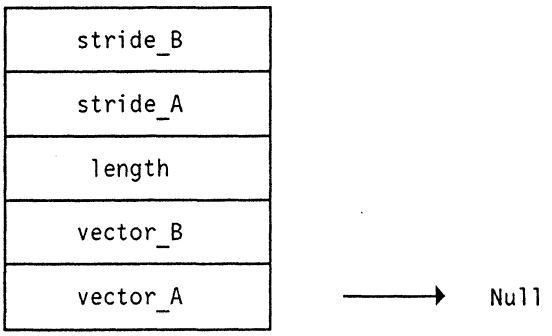
```
for i in {1 to length} do A(i) := A(i) * B(i)
```

The specific type requirements [scalar or vector; width] are indicated for each operator between brackets at the right of the function definitions.

Dyadic Vector Operators

These operators act on two vectors. The first five copy a function of B into A; the rest replace A by a function of A and B.

Top-of-stack transformation:



CPV <copy vector>

```
A := B                [B may be scalar; wB = wA]
```

CPVN <copy vector negated>

```
A := - B              [B may be scalar; wB = wA]
```

CPVA <copy vector absolute>

```
A := abs(B)           [B may be scalar; wB = wA]
```

CPVS <copy vector single>

```
A := single (B)       [B must be vector; wA = 1]
```

Each element of B is converted to single precision, as defined for the SNGL operator.

CPVD <copy vector double>

```
A := double (B)       [B must be vector; wA = 2]
```

Each element of B is converted to double precision, as defined for the XTND operator.

VPV <vector plus vector>

```
A := A + B            [B may be scalar; wB <= wA]
```

VMV <vector minus vector>

```
A := A - B            [B may be scalar; wB <= wA]
```

VMVN <vector minus vector negated>

```
A := -A + B           [B may be scalar; wB <= wA]
```

VTV <vector times vector>

```
A := A * B            [B may be scalar; wB <= wA]
```

VOV <vector over (divided by) vector>

```
A := A / B            [B may be scalar; wB <= wA]
```

VUV <vector under (divided into) vector>

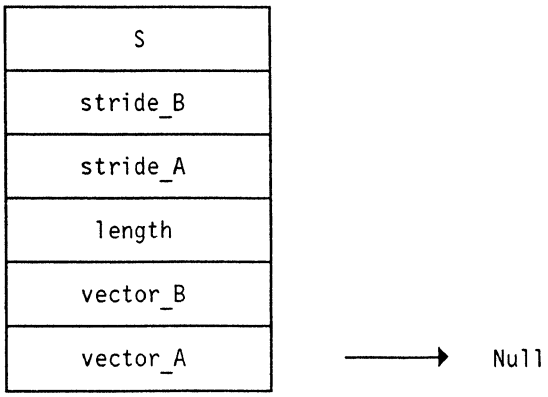
```
A := A \ B = B / A    [B may be scalar; wB <= wA]
```

Dyadic Vector Operators with Scalar

These operators act on two vectors and a scalar. S, the scalar, is an operand argument; its width is called wS.

The first two operators assign to A a function of B and S; the other operators of this group assign to A a function of A, B, and S.

Top-of-stack transformation:



VPS <vector plus scalar>

```
A := B + S          [B must be vector; wB <= wS = wA]
```

VTS <vector times scalar>

```
A := B * S          [B must be vector; wB <= wS = wA]
```

VPVS <vector plus vector times scalar>

```
A := A + B * S      [B must be vector; wB <= wS = wA]
```

VSPV <vector times scalar plus vector>

```
A := A * S + B      [B may be scalar; wB <= wS = wA]
```

Scatter/Gather Operators

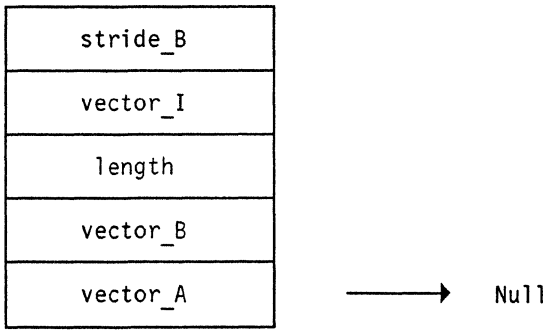
These operators accept an IndexedSingleDD instead of one of the stride arguments. The descriptor, called vector_I, designates a vector of virtual index values; these are used to index the vector_A or vector_B argument to access an arbitrary sequence of elements. The elements of I are accessed sequentially with stride +1; each element must be an integer with magnitude less than MaxLen.

The operation is undefined if I overlaps A.

SCAT <scatter>

```
for i in {1 to length} do     A[I(i)] := B(i)       [B may be scalar; wB = wA]
```

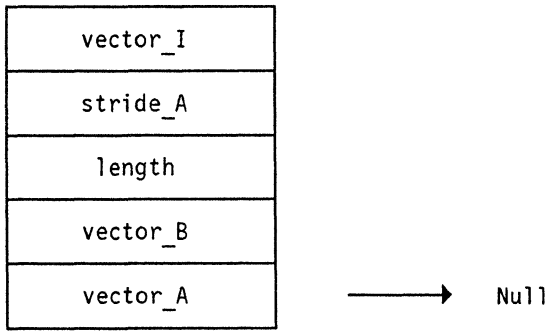
Top-of-stack transformation:



GATH <gather>

```
for i in {1 to length} do     A(i) := B[I(i)]       [B must be vector; wB = wA]
```

Top-of-stack transformation:

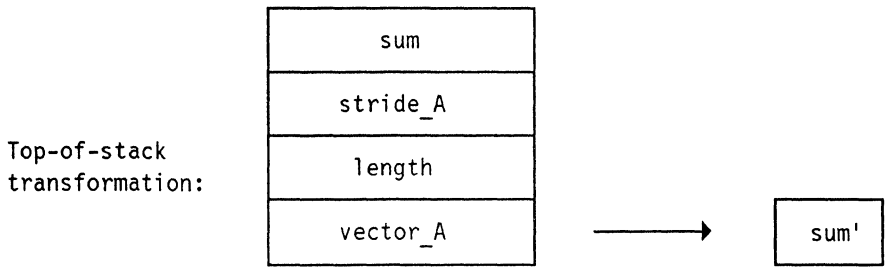


Reduction Operators

These operators read one or two vectors and produce a scalar result. The vector arguments must be IndexedWordDDs; they need not be write enabled.

Linear Sum

These operators sum the elements in a single vector.



The sum argument must be an operand with width greater than or equal to wA; the arithmetic is performed in double precision if sum is double.

SUM <summation over vector>

```
for i = 1 to length do    sum := sum + A(i)
```

SUMA <summation over vector, absolute>

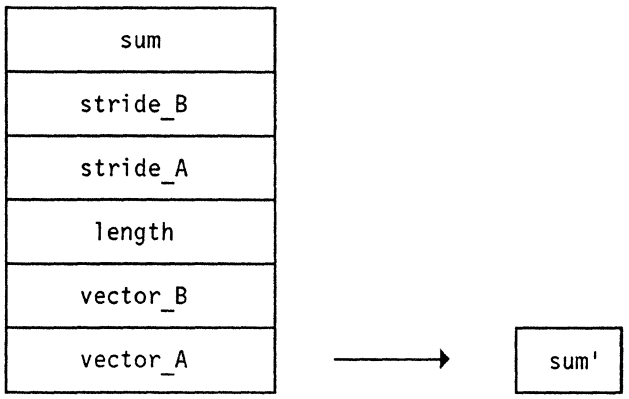
```
for i = 1 to length do    sum := sum + abs(A(i))
```

Quadratic Sum

These operators compute the inner product of two vectors:

```
for i = 1 to length do    sum := sum + A(i) * B(i).
```

Top-of-stack transformation:



B must be a vector.

DOT <dot (inner) product - single>

The sum argument must be a single precision operand; vector_A and vector_B must be IndexedSingleDDs. Multiplication and addition are performed in single precision.

DOTX <dot (inner) product - extended>

The sum argument must be a double precision operand. Vector_A and vector_B must be IndexedWordDDs; wA and wB are independent. All arithmetic is double precision; multiplication is performed as in the MULX operator.

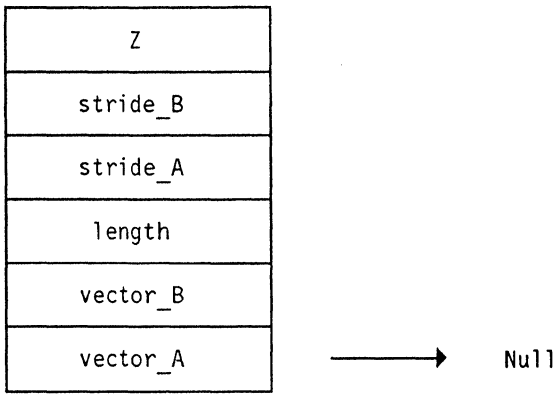
Recurrence Operators

Each of these operators iteratively computes a scalar function in which, at each iteration, the previous scalar value enters into the computation. The first operator iteratively assigns values to the elements of vector A; the others return the final scalar value.

SEQ <sequential recurrence>

```
for i = 1 to length do     A(i) := Z := Z + B(i);
```

Top-of-stack transformation:

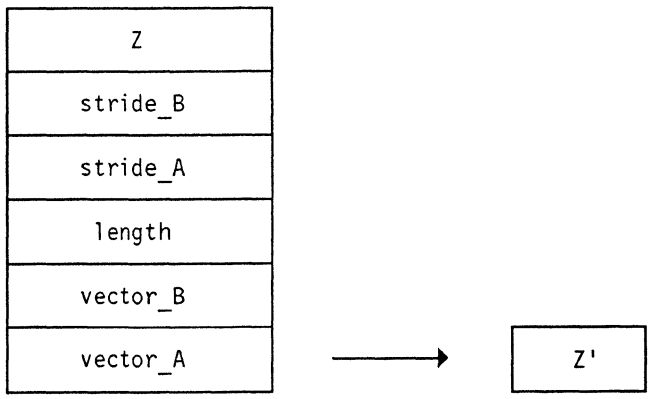


B may be a scalar or vector; wB must be less than or equal wA. Z must be an operand with width equal wA. Vector_A must be write enabled.

POLY <polynomial recurrence>

```
for i = 1 to length do          Z := A(i) + B(i) * Z
```

Top-of-stack transformation:

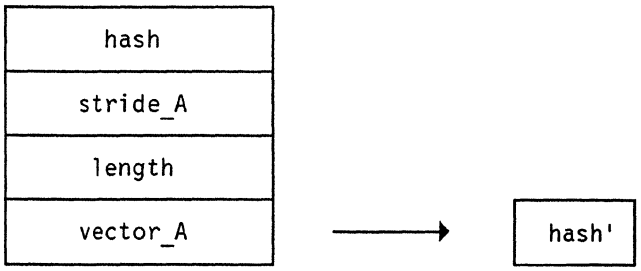


B may be a scalar or vector; wB and wA are independent. Z must be an operand with width greater than or equal to max(wA,wB); the computation is performed and the result returned with the width of Z.

CHEK <compute check-hash>

```
for i = 1 to length do     hash := (hash EQV A(i)).[46:48]
```

Top-of-stack transformation:

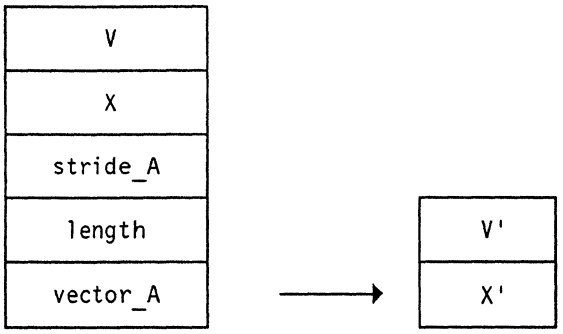


Vector_A must be an IndexedSingleDD; hash must be a single precision operand. Each iteration consists of a one-bit left circular shift of the bit-vector logical equivalence of the accumulator and the next vector element.

Search Operators

These operators find the virtual index of an extremum (maximum, minimum, or maximum absolute value) in a vector.

Top-of-stack transformation:



X must be a single precision operand. V must be an operand with width equal to wA.

The operators iteratively compare the vector elements with V; whenever an element of A is more extreme than V, it replaces V and its virtual index replaces X. The final values of V and X are left as the results of the operator.

If the same extreme value occurs more than once in A, the first one encountered is reported. If no element in A is more extreme than V, the original arguments V and X are unchanged; this is the only case in which the value of argument X is significant.

Note that the index returned (when changed by the operator) is a virtual index into the segment. Thus, if the target array is contained within a larger virtual segment (such as a FORTRAN common block), the offset of the array into the segment must be subtracted from the operator result to derive an array-relative index. Furthermore, the virtual index is defined in words, whether vector_A is an IndexedSingleDD or IndexedDoubleDD.

The function VirtualIndex(A(i)) in the following operator definitions is the index (into the virtual segment containing A) of the item examined in the ith iteration of the operator. It is equivalent to the virtual index that would be reported by the GINX operator applied to an IndexedWordDD created by applying the INDX operator to descriptor vector_A with index (i-1)*stride_A.

FMX <find maximum>

```
for i = 1 to length do
 if A(i) > V then V,X := A(i),VirtualIndex(A(i))
```

FMN <find minimum>

```
for i = 1 to length do
 if A(i) < V then V,X := A(i),VirtualIndex(A(i))
```

FMXA <find maximum absolute value>

```
for i = 1 to length do
 if abs(A(i)) > V then V,X := abs(A(i)),VirtualIndex(A(i))
```

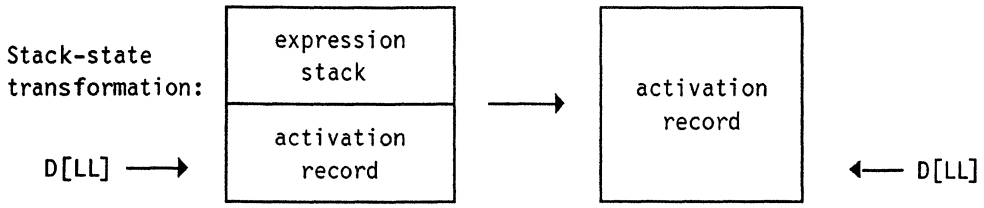
Miscellaneous Operators

NOOP <no operation>

No action is performed.

PUSH <push working stack onto activation record>

PUSH adds the contents of the expression stack to the topmost activation record, leaving a null expression stack. The operation is invalid when a procedure entry code sequence is being executed (there is an incipient AR at the top of the stack).



NVLD <invalid operator>

An Invalid Operator interrupt is unconditionally generated.

VARI <introduce variant operator>

The VARI operator may be considered a primary operator that causes the next code syllable to be interpreted as a variant operator. The two operator syllables are tightly bound, in that no Asynchronous or External interrupt can occur between the VARI and the introduced operator and any RCW that designates a variant operator must point to the VARI.

Program Restrictions Due to Hidden State

In addition to visible state, which is accessible directly via various operators, a processor maintains other state to facilitate efficient execution. For the most part, such state is beyond the scope of the functional specification. However, some restrictions on software are necessary to ensure that the processor captured state is consistent with the visible state, particularly memory contents.

For example, a program cannot use a table edit sequence to alter the edit table in use for that operation. It is likewise undefined for a translate or set membership operator to modify as a destination the table it is using. (However, any store to the table performed by the same process prior to the translate or set membership operator must be effective.)

The processor is permitted to capture for each current address couple the contents of the word designated for the address couple. If the address designates a reference, the processor may capture the contents of the word designated by the reference chain.

The association remains valid until the addressing environment is changed so as to redefine the address couple, unless a synchronizing operator (RDLK or LOK) is executed or the word is explicitly written via the address couple, an SIRW, or an IndexedWordDD. (The processor is not required to recognize writing via an ASRW.) In addition to single word values, the processor can capture a double precision operand:

- When an address couple directly designates a double precision operand, the processor can associate the double operand value with the address couple. The processor is not required to recognize modification of the second word; thus the following sequence is not permitted:

```
VALC  lambda, delta
DUPL
NAMC  lambda, delta+1
STOD
VALC  lambda, delta        <------  (produces undefined result)
```

For more information on program restrictions for restricted operators see Appendix E.

Appendix A
Alphabetical Operator Code List

Table A-1 lists all operators alphabetically by mnemonic. The operator mnemonic and name are followed by the hexadecimal encoding, operator type, and parameters, if any. For operators that are not valid on all A Series systems, information concerning compatibility is provided. Operators that are restricted are also denoted as such.

The operator type is indicated with the following notation:

p	primary operator
v	variant operator
e	edit mode operator (same encoding in table and single edit)
se	edit mode operator (single edit only)
te	edit mode operator (table edit only)

Table A-1. Operators, Alphabetical Listing

Mnemonic	Operator Name	Hexadecimal	Operator Type	Code Stream Parameters	Level/Feature
ADD	Add	80	p		
AMAX	Arithmetic maximum	958A	v		
AMIN	Arithmetic minimum	9588	v		
ASRT	Assert	9580	v	interrupt_ code:8	
AVER	Aver	958F	v		MCP/AS (Extended)
BCD	Binary convert to decimal	9577	v	N:8	
BMS	Bounded masked search for equal	95A9	v		MCP/AS (Extended) plus some others
BRFL	Branch false	A0	p	op_psi:3 op_pwi:13	

continued

Mnemonic	Operator Name	Hexadecimal	Operator Type	Code Stream Parameters	Level/Feature
BRST	Bit reset	9E	p	Db:8	
BRTR	Branch true	A1	p	op_psi:3 op_pwi:13	
BRUN	Branch unconditional	A2	p	op_psi:3 op_pwi:13	
BSET	Bit set	96	p	Db:8	
CBON	Count binary ones	95BB	v		
CEQD	Compare characters equal delete	F4	p		
CEQU	Compare characters equal update	FC	p		
CGED	Compare characters greater or equal delete	F1	p		
CGEU	Compare characters greater or equal update	F9	p		
CGTD	Compare characters greater delete	F2	p		
CGTU	Compare characters greater update	FA	p		
CHEK	Compute check-hash	95C8	v		MCP/AS (Extended)
CHSN	Change sign	8E	p		
CLED	Compare characters less or equal delete	F3	p		
CLEU	Compare characters less or equal update	FB	p		

continued

Mnemonic	Operator Name	Hexadecimal	Operator Type	Code Stream Parameters	Level/Feature
CLSD	Compare characters less delete	F0	p		
CLSU	Compare characters less update	F8	p		
CNED	Compare characters not equal delete	F5	p		
CNEU	Compare characters not equal update	FD	p		
CPV	Copy vector	95E0	v		MCP/AS (Extended)
CPVA	Copy vector absolute	95E2	v		MCP/AS (Extended)
CPVD	Copy vector double	95E4	v		MCP/AS (Extended)
CPVN	Copy vector negated	95E1	v		MCP/AS (Extended)
CPVS	Copy vector single	95E3	v		MCP/AS (Extended)
CSCP	Communicate with System Control Processor	959F	v		Unimplemented Op Restricted
DBCD	Dynamic binary convert to decimal	957F	v		
DBFL	Dynamic branch false	A8	p		
DBRS	Dynamic bit reset	9F	p		
DBST	Dynamic bit set	97	p		
DBTR	Dynamic branch true	A9	p		
DBUN	Dynamic branch unconditional	AA	p		

continued

Mnemonic	Operator Name	Hexadecimal	Operator Type	Code Stream Parameters	Level/Feature
DEXI	Disable external interrupts	9547	v		
DFTR	Dynamic field transfer	99	p		
DINS	Dynamic field insert	9D	p		
DISO	Dynamic field isolate	9B	p		
DIVD	Divide	83	p		
DLET	Delete top of stack	B5	p		
DOT	Dot (inner) product-- single	95C4	v		MCP/AS (Extended)
DOTX	Dot (inner) product-- extended	95C5	v		MCP/AS (Extended)
DRNT	Dynamic range test	9583	v		
DSLF	Dynamic scale left	C1	p		
DSRF	Dynamic scale right final	C7	p		
DSRR	Dynamic scale right rounded	C9	p		
DSRS	Dynamic scale right save	C5	p		
DSRT	Dynamic scale right truncate	C3	p		
DUPL	Duplicate top of stack	B7	p		
EEXI	Enable external interrupts	9546	v		
ENDE	End edit	DE	te		

continued

Mnemonic	Operator Name	Hexadecimal	Operator Type	Code Stream Parameters	Level/Feature
ENDF	End float	D5	te	MinusChar:8 PlusChar:8	
ENTR	Enter	AB	p		
EQUL	Equal to	8C	p		
EVAL	Evaluate	AC	p		
EXCH	Exchange top of stack	B6	p		
EXIT	Exit	A3	p		
EXPU	Execute single edit operator, single pointer update	DD	p		
EXSD	Execute single edit operator delete	D2	p		
EXSU	Execute single edit operator update	DA	p		
FLTR	Field transfer	98	p	Db:8 Sb:8 Len:8	
FMN	Find minimum	95CA	v		MCP/AS (Extended)
FMX	Find maximum	95C9	v		MCP/AS (Extended)
FMXA	Find maximum absolute value	95CB	v		MCP/AS (Extended)
GATH	Gather	95C1	v		MCP/AS (Extended)
GESZ	Get element_size	95CF	v		MCP/AS (Extended)
GINX	Get index	95CE	v		MCP/AS (Extended)
GLEN	Get length	95CD	v		MCP/AS (Extended)
GREQ	Greater than or equal to	89	p		
GRTR	Greater than	8A	p		

continued

Mnemonic	Operator Name	Hexadecimal	Operator Type	Code Stream Parameters	Level/Feature
HALT	Conditional processor halt	95DF	v		Restricted
ICLD	Input convert left-signed delete	9575	v		
ICRD	Input convert right-signed delete	9576	v		
ICUD	Input convert unsigned delete	A4	p		
ICVD	Input convert delete	CA	p		
ICVU	Input convert update	CB	p		
IDIV	Integer divide	84	p		
IMKS	Insert mark stack	CF	p		
INDX	Index	A6	p		
INOP	Insert overpunch	D8	e		
INSC	Insert conditional	DD	te	Length:8 ZeroChar:8 NonZeroChar:8	
INSG	Insert display sign	D9	e	MinusChar:8 PlusChar:8	
INSR	Field Insert	9C	p	Db:8 Len:8	
INSU	Insert unconditional	DC	se	Char:8	
INSU	Insert unconditional	DC	te	Length:8 Char:8	
INXA	Index via address couple parameter	E7	p	lambda:4 delta:12	Phased out
ISOL	Field isolate	9A	p	Sb:8 Len:8	
JOIN	Join two operands into double	9542	v		

continued

Mnemonic	Operator Name	Hexadecimal	Operator Type	Code Stream Parameters	Level/Feature
LAND	Logical and	90	p		
LEQV	Logical equivalence	93	p		
LESS	Less than	88	p		
LNMC	Long name call	958C	v	lambda:4 delta:12	
LNOT	Logical not	92	p		
LOAD	Load	BD	p		
LODT	Load transparent	BC	p		
LODT	Load transparent	95BC	v		
LOG2	Leading one test	958B	v		
LOK	Lock interlock	95B0	v		MCP/AS (Extended) plus some others
LOR	Logical or	91	p		
LPOW	Load POW	954D	v		Restricted MCP/AS (Extended)
LSEQ	Less than or equal to	8B	p		
LT8	Insert 8 bit literal	B2	p	constant:8	
LT16	Insert 16 bit literal	B3	p	constant:16	
LT48	Insert 48 bit literal	BE	p	word-aligned constant:48	
LVLC	Long value call	958D	v	lambda:4 delta:12	
MCHR	Move characters	D7	se		
MCHR	Move characters	D7	te	Length:8	
MFLT	Move with float	D1	se	ZeroChar:8 MinusChar:8 PlusChar:8	

continued

Mnemonic	Operator Name	Hexadecimal	Operator Type	Code Stream Parameters	Level/Feature
MFLT	Move with float	D1	te	Length:8 ZeroChar:8 MinusChar:8 PlusChar:8	
MINS	Move with insert	D0	se	ZeroChar:8	
MINS	Move with insert	D0	te	Length:8 ZeroChar:8	
MKSN	Mark-stack bound to name-call	DF	p		
MKST	Mark stack	AE	p		
MPCW	Make PCW	BF	p	word-aligned SkeletonPCW: 48	
MULT	Multiply	82	p		
MULX	Extended multiply	8F	p		
MVNU	Move numeric unconditional	D6	se		
MVNU	Move numeric unconditional	D6	te	Length:8	
NAMC	Name call	40-7F	p	variable-fence address couple:14 (two syllables total)	
NEQL	Not equal to	8D	p		
NOOP	No operation	FE	p		
NOOP	No operation	95FE	v		
NORM	Normalize	958E	v		
NORX	Disable reindex of reference	957C	v		MCP/AS (Extended)
NOWR	Disable write via reference	957D	v		MCP/AS (Extended)

continued

Mnemonic	Operator Name	Hexadecimal	Operator Type	Code Stream Parameters	Level/Feature
NTGD	Integerize double precision rounded	9587	v		
NTGR	Integerize rounded	87	p		
NTIA	Integerize truncated	86	p		
NTTD	Integerize double precision truncated	9586	v		
NVLD	Invalid operator	FF	p		
NVLD	Invalid operator	95FF	v		
NXLN	Index and load name	A5	p		
NXLV	Index and load value	AD	p		
NXVA	Index and load value via address couple parameter	EF	p	lambda:4 delta:12	Phased out
OCRX	Occurs index	9585	v		
ONE	Insert literal one	B1	p		
OVRD	Overwrite delete	BA	p		
OVRN	Overwrite non-delete	BB	p		
PACD	Pack delete	D1	p		
PACU	Pack update	D9	p		
PKLD	Pack left-signed delete	9573	v		
PKRD	Pack right-signed delete	9574	v		
PKUD	Pack unsigned delete	9572	v		

continued

Mnemonic	Operator Name	Hexadecimal	Operator Type	Code Stream Parameters	Level/Feature
POLY	Polynomial recurrence	95C7	v		MCP/AS (Extended)
PUSH	Push working stack onto activation record	B4	p		
RASD	Read ASD-table word	95B1	v		Restricted
RAST	RASD trapping synonym	95D6	v		Restricted special mode
RDIV	Remainder divide	85	p		
RDLK	Read and lock	95BA	v		
RETN	Return	A7	p		
REXI	Return/exit from interrupt	95F7	v		Restricted MCP/AS (Extended)
RNGT	Range test	9582	v	l:8 h:8	
ROFF	Read and reset overflow flip-flop	D7	p		
RPRR	Read processor register	95B8	v		
RSDN	Rotate stack down	95B7	v		
RSNR	Read SNR	9581	v		
RSQR	Real square root	9550	v		MCP/AS (Extended)
RSTF	Reset float flip-flop	D4	e		
RSUP	Rotate stack up	95B6	v		
RTAG	Read tag	95B5	v		
RTFF	Read true-false flip-flop	DE	p		
RTN2	Return two items	95AE	v		MCP/AS (Extended)

continued

Mnemonic	Operator Name	Hexadecimal	Operator Type	Code Stream Parameters	Level/Feature
RTOD	Read time of day clock	95A7	v		
RUNI	Indicate running	9541	v		Restricted
SAME	Logical equality	94	p		
SCAT	Scatter	95C0	v		MCP/AS (Extended)
SCLF	Scale left	C0	p	ScaleFactor:8	
SCRF	Scale right final	C6	p	ScaleFactor:8	
SCRR	Scale right rounded	C8	p	ScaleFactor:8	
SCRS	Scale right save	C4	p	ScaleFactor:8	
SCRT	Scale right truncate	C2	p	ScaleFactor:8	
SEDW	Set element_size to double words	95A3	v		MCP/AS (Extended)
SEQ	Sequential recurrence	95C6	v		MCP/AS (Extended)
SEQD	Scan while equal delete	95F4	v		
SEQU	Scan while equal update	95FC	v		
SESW	Set element_size to single words	95A2	v		MCP/AS (Extended)
SE4C	Set element_size to hex characters	95A0	v		MCP/AS (Extended)
SE8C	Set element_size to EBCDIC characters	95A1	v		MCP/AS (Extended)
SFDC	Skip forward destination characters	DA	se		
SFDC	Skip forward destination characters	DA	te	Length:8	

continued

Mnemonic	Operator Name	Hexadecimal	Operator Type	Code Stream Parameters	Level/Feature
SFSC	Skip forward source characters	D2	se		
SFSC	Skip forward source characters	D2	te	Length:8	
SGED	Scan while greater or equal delete	95F1	v		
SGEU	Scan while greater or equal update	95F9	v		
SGTD	Scan while greater delete	95F2	v		
SGTU	Scan while greater update	95FA	v		
SHOW	Primitive display	95DE	v		Unimplemented Op Restricted
SISO	String isolate	D5	p		
SLED	Scan while less or equal delete	95F3	v		
SLEU	Scan while less or equal update	95FB	v		
SLSD	Scan while less delete	95F0	v		
SLSU	Scan while less update	95F8	v		
SNED	Scan while not equal delete	95F5	v		
SNEU	Scan while not equal update	95FD	v		
SNGL	Set to single precision rounded	CD	p		
SNGT	Set to single precision truncated	CC	p		

continued

Mnemonic	Operator Name	Hexadecimal	Operator Type	Code Stream Parameters	Level/Feature
SPLT	Split operand into two singles	9543	v		
SPRR	Set processor register	95B9	v		Restricted
SRCH	Masked search for equal	95BE	v		
SRDC	Skip reverse destination characters	DB	se		
SRDC	Skip reverse destination characters	DB	te	Length:8	
SRSC	Skip reverse source characters	D3	se		
SRSC	Skip reverse source characters	D3	te	Length:8	
STAD	Store delete via address couple parameter	F6	p	lambda:4 delta:12	Phased out
STAG	Set tag	95B4	v		
STAN	Store nondelete via address couple parameter	F7	p	lambda:4 delta:12	Phased out
STFF	Stuff	AF	p		
STOD	Store delete	B8	p		
STON	Store nondelete	B9	p		
STOP	Unconditional processor halt	95BF	v		
SUBT	Subtract	81	p		
SUM	Summation over vector	95C2	v		MCP/AS (Extended)

continued

Mnemonic	Operator Name	Hexadecimal	Operator Type	Code Stream Parameters	Level/Feature
SUMA	Summation over vector absolute	95C3	v		MCP/AS (Extended)
SWFD	Scan while false delete	95D4	v		
SWFU	Scan while false update	95DC	v		
SWTD	Scan while true delete	95D5	v		
SWTU	Scan while true update	95DD	v		
SXSN	Set external sign flip-flop	D6	p		
TEED	Table enter edit delete	D0	p		
TEEU	Table enter edit update	D8	p		
TEQD	Transfer while equal delete	E4	p		
TEQU	Transfer while equal update	EC	p		
TGED	Transfer while greater or equal delete	E1	p		
TGEU	Transfer while greater or equal update	E9	p		
TGTD	Transfer while greater delete	E2	p		
TGTU	Transfer while greater update	EA	p		
TLED	Transfer while less or equal delete	E3	p		

continued

Mnemonic	Operator Name	Hexadecimal	Operator Type	Code Stream Parameters	Level/Feature
TLEU	Transfer while less or equal update	EB	p		
TLSD	Transfer while less delete	E0	p		
TLSU	Transfer while less update	E8	p		
TNED	Transfer while not equal delete	E5	p		
TNEU	Transfer while not equal update	ED	p		
TRNS	Translate	95D7	v		
TUND	Transfer characters unconditional delete	E6	p		
TUNU	Transfer characters unconditional update	EE	p		
TWFD	Transfer while false delete	95D2	v		
TWFU	Transfer while false update	95DA	v		
TWOD	Transfer words overwrite delete	D4	p		Restricted
TWSD	Transfer words delete	D3	p		
TWSU	Transfer words update	DB	p		
TWTD	Transfer while true delete	95D3	v		
TWTU	Transfer while true update	95DB	v		
UNLK	Unlock interlock	95B2	v		MCP/AS (Extended) plus some others

continued

Mnemonic	Operator Name	Hexadecimal	Operator Type	Code Stream Parameters	Level/Feature
UPLD	Unpack left-signed delete	9570	v		
UPLU	Unpack left-signed update	9578	v		
UPRD	Unpack right-signed delete	9571	v		
UPRU	Unpack right-signed update	9579	v		
UPUD	Unpack unsigned delete	95D1	v		
UPUU	Unpack unsigned update	95D9	v		
USND	Unpack signed delete	95D0	v		
USNU	Unpack signed update	95D8	v		
VALC	Value call	00-3F	p	variable-fence address couple:14 (two syllables total)	
VARI	Introduce variant operator	95	p		
VMV	Vector minus vector	95E6	v		MCP/AS (Extended)
VMVN	Vector minus vector negated	95E7	v		MCP/AS (Extended)
VOV	Vector over (divided by) vector	95E9	v		MCP/AS (Extended)
VPS	Vector plus scalar	95EB	v		MCP/AS (Extended)
VPV	Vector plus vector	95E5	v		MCP/AS (Extended)
VPVS	Vector plus vector times scalar	95ED	v		MCP/AS (Extended)
VSPV	Vector times scalar plus vector	95EE	v		MCP/AS (Extended)

continued

Mnemonic	Operator Name	Hexadecimal	Operator Type	Code Stream Parameters	Level/Feature
VTS	Vector times scalar	95EC	v		MCP/AS (Extended)
VTV	Vector times vector	95E8	v		MCP/AS (Extended)
VUV	Vector under (divided into) vector	95EA	v		MCP/AS (Extended)
WASD	Write ASD-table word	95B3	v		Restricted
WAST	WASD trapping synonym	95A6	v		Restricted special mode
WATI	Read machine identification	95A4	v		Restricted
WHOI	Read proc_id	954E	v		Restricted
WTOD	Write time of day clock	9549	v		Restricted Unimplemented Op
XFNC	External function call	959E	v		Restricted Not defined for many systems
XTND	Set to double precision	CE	p		
ZERO	Insert literal zero	B0	p		
ZIC	Zero Interrupt_Count	9540	v		Restricted

Appendix B
Boolean Accumulator X-Ref

The operators that interact with the four Boolean Accumulators are listed with the type of interaction.

TFFF (true/false flip flop)

left in undefined state by:

TUND/TUNU, PKUD, ICUD, ICLD, ICRD, ICVD/ICVU

set by:

character relational operators:

SGTD/SGTU, SGED/SGEU, SEQD/SEQU, SNED/SNEU, SLED/SLEU, SLSD/SLSU, TGTD/TGTU, TGED/TGEU, TEQD/TEQU, TNED/TNEU, TLED/TLEU, TLSD/TLSU

character sequence compare delete operators:

CGTD, CGED, CEQD, CNED, CLED, CLSD

character sequence compare update operators (not after relation determined):

CGTU, CGEU, CEQU, CNEU, CLEU, CLSU

character set membership operators:

SWTD/SWTU, SWFD/SWFU, TWTD/TWTU, TWFD/TWFU

pack operators (except PKUD):

PKLD, PKRD, PACD/PACU

read by:

RTFF

OFFF (overflow flip flop)

conditionally set to 1 (true) by:

SCLF/DSLF, SCRF/DSRF, BCD/DBCD

read and reset to 0 (false) by:

ROFF

EXTF (external sign flip flop)

left in undefined state by:

PKUD, ICUD, ICLD, ICRD, ICVD/ICVU, SCRS, DSRS

set by:

SCRF/DSRF, BCD/DBCD, SXSN, PKLD, PKRD, PACD/PACU

read by:

unpack signed operators:

UPLD/UPLU, UPRD/UPRU, USND/USNU

edit sign operators:

INOP, INSG, MFLT, ENDF

FLTF (float flip flop)

reset to 0 (false) by:

TEED/TEEU, EXSD/EXSU in initial mode and by RSTF

read and conditionally set to 1 (true) by:

MINS, MFLT

read and reset to 0 (false) by:

ENDF

TFFF, OFFF, and EXTF

saved in the stack frame linkage by:

enter operations

restored from that linkage by:

exit operations

FLTF

preserved as restart information at service interrupts from:

edit operators

restored by:

enter-edit operators in restart mode

Appendix C
Operator Encoding

The processor has three modes of interpretation of a one-syllable operator code.

1. Primary mode is the default interpretation of one-syllable operating code.
2. Variant mode is enabled by interpretation of the primary mode operator encoding VARI (hex 95). VARI causes the processor to interpret the next syllable as a variant mode operator. Following execution of this operator, the processor reverts to primary mode. For convenience, variant operator encodings are written hex 95xx, where xx is the one-syllable variant operator code.
3. Edit mode is enabled by interpretation of the primary mode operators TEED, TEEU, EXSD, EXSU, and EXPU (hex D0, D8, D2, DA, and DD, respectively). TEED and TEEU cause the processor to interpret the syllables in an external table as edit mode operators, each of which is followed by associated parameters. The processor returns to primary mode following interpretation of ENDE (hex DE) operator in the table. EXSD, EXSU, and EXPU cause the processor to interpret the next syllable as an edit mode operator followed by associated parameters. Following execution of the edit mode operator, the processor returns to primary mode interpretation.

The following charts show for each interpretation mode the operator associated with each syllable encoding. Unflagged mnemonics are unrestricted; flag characters denote obsolescent, obsolete or restricted operators:

- \- obsolescent or obsolete
- \+ partially restricted
- \# restricted

Primary Mode Operators

	0	1	2	3	4	5	6	7
8	ADD	SUBT	MULT	DIVD	IDIV	RDIV	NTIA	NTGR
9	LAND	LOR	LNOT	LEQV	SAME	VARI	BSET	DBST
A	BRFL	BRTR	BRUN	EXIT	ICUD	NXLN	INDX	RETN
B	ZERO	ONE	LT8	LT16	PUSH	DLET	EXCH	DUPL
C	-SCLF	-DSLF	SCRT	DSRT	-SCRS	-DSRS	-SCRF	-DSRF
D	TEED	-PACD	EXSD	TWSD	#TWOD	SISO	SXSN	ROFF
E	TLSD	TGED	TGTD	TLED	TEQD	TNED	TUND	-INXA
F	CLSD	CGED	CGTD	CLED	CEQD	CNED	-STAD	-STAN

Primary encodings are 00-3F are VALC; primary encodings 40-7F are NAMC.

8	9	A	B	C	D	E	F	
LESS	GREQ	GRTR	LSEQ	EQUL	NEQL	CHSN	MULX	8
FLTR	DFTR	ISOL	DISO	INSR	DINS	BRST	DBRS	9
DBFL	DBTR	DBUN	ENTR	EVAL	NXLV	MKST	STFF	A
STOD	STON	+OVRD	+OVRN	+LODT	LOAD	LT48	MPCW	B
SCRR	DSRR	-ICVD	-ICVU	SNGT	SNGL	XTND	IMKS	C
TEEU	-PACU	EXSU	TWSU		EXPU	RTFF	MKSN	D
TLSU	TGEU	TGTU	TLEU	TEQU	TNEU	TUNU	-NXVA	E
-CLSU	-CGEU	-CGTU	-CLEU	-CEQU	-CNEU	NOOP	NVLD	F

Variant Mode Operators

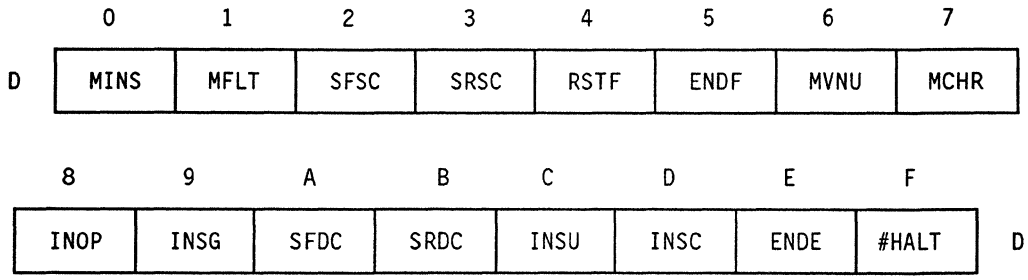
	0	1	2	3	4	5	6	7
3								
4	#ZIC	#RUNI	JOIN	SPLT			+EEXI	+DEXI
5	RSQR							
6								
7	UPLD	UPRD	PKUD	PKLD	PKRD	ICLD	ICRD	BCD
8	ASRT	RSNR	RNGT	DRNT		OCRX	NTTD	NTGD
9								
A	SE4C	SE8C	SESW	SEDW	#WATI		#WAST	RTOD
B	LOK	#RASD	UNLK	#WASD	+STAG	RTAG	RSUP	RSDN
C	SCAT	GATH	SUM	SUMA	DOT	DOTX	SEQ	POLY
D	-USND	UPUD	TWFD	TWTD	SWFD	SWTD	#RAST	TRNS
E	CPV	CPVN	CPVA	CPVS	CPVD	VPV	VMV	VMVN
F	SLSD	SGED	SGTD	SLED	SEQD	SNED		#REXI

Variant encodings 00-2F are undefined.

8	9	A	B	C	D	E	F	
								3
	#WTOD				#LPOW	#WHOI		4
								5
								6
UPLU	UPRU			NORX	NOWR		DBCD	7
AMIN		AMAX	LOG2	LNMC	LVLC	NORM	AVER	8
						#XFNC	#CSCP	9
	BMS					RTN2		A
+RPRR	#SPRR	+RDLK	CBON	+LODT		SRCH	#STOP	B
CHEK	FMX	FMN	FMXA		GLEN	GINX	GESZ	C
-USNU	UPUU	TWFU	TWTU	SWFU	SWTU	#SHOW	#HALT	D
VTV	VOV	VUV	VPS	VTS	VPVS	VSPV		E
SLSU	SGEU	SGTU	SLEU	SEQU	SNEU	NOOP	NVLD	F

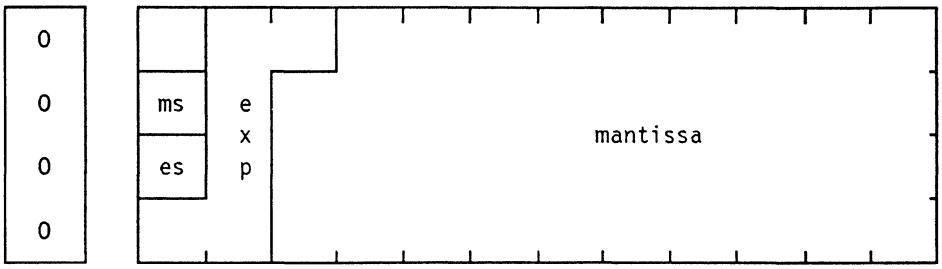
Edit Mode Operators

All edit operator encodings except D0-DF are undefined.



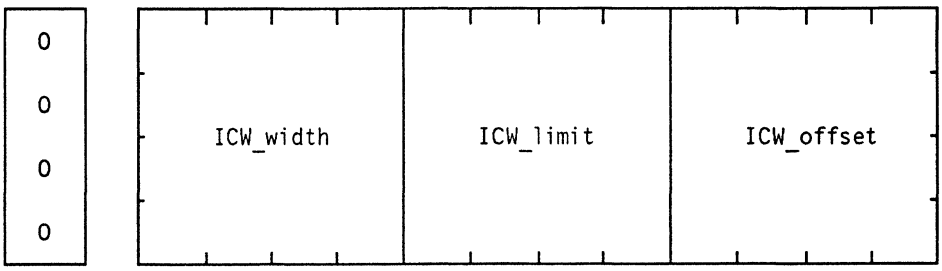
Appendix D
Data Type Formats

Single Precision, numeric interpretation



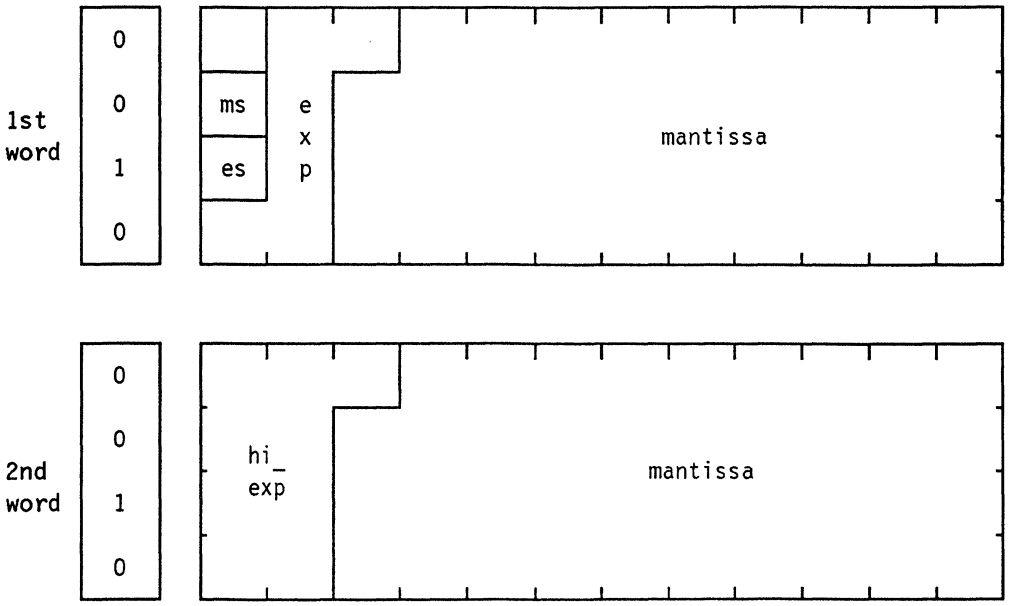
mant_sign	[46: 1]	Mantissa sign (0=positive, 1=negative)
exp_sign	[45: 1]	Exponent sign (0=positive, 1=negative)
exponent	[44: 6]	The power of eight to which the mantissa is scaled
mantissa	[38:39]	The magnitude of the number before scaling

Single Precision, ICW (Index Control Word) interpretation



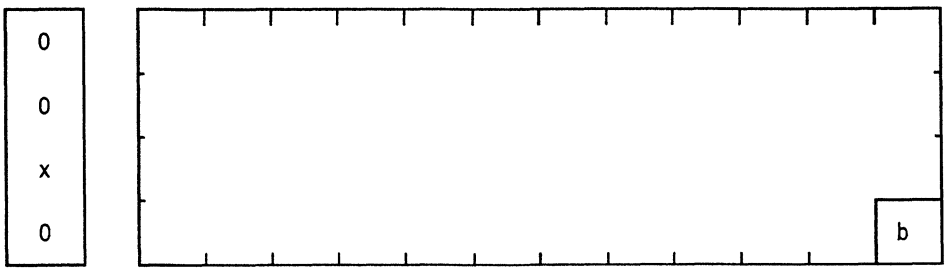
ICW_width	[47:16]	The number of units in an element in the sequence
ICW_limit	[31:16]	The number of elements in the sequence
ICW_offset	[15:16]	The offset from the base of the record to the start of the sequence

Double Precision, numeric interpretation



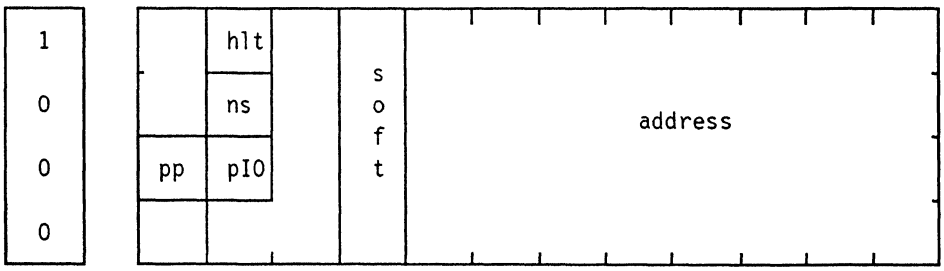

```
1st word:
mant_sign       [46: 1]  Mantissa sign (0=positive, 1=negative)
exp_sign        [45: 1]  Exponent sign (0=positive, 1=negative)
exponent        [44: 6]  The low order 6 bits of the exponent
mantissa        [38:39]  The integral portion of the mantissa

2nd word:
hi_order_exp    [47: 9]  The high order 9 bits of the exponent
mantissa        [38:39]  The fractional portion of the mantissa
```

Single or Double Precision, Boolean interpretation



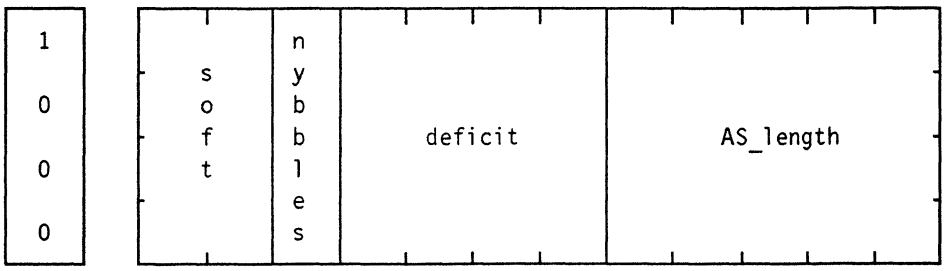

```
tag             (0: single precision,
                 2: double precision (2nd word ignored))
[ 0: 1]  Boolean value (0=false, 1=true)
```

Actual Segment Descriptor, First Word (ASD1)



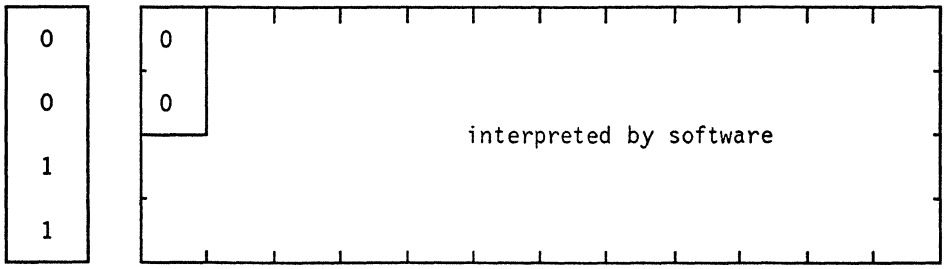
	[47: 2]	Reserved for co-existence
present_to_proc	[45: 1]	Segment-is-present indicator for non-I/O access (0=absent, 1=present)
	[44: 1]	Reserved for co-existence
hlt	[43: 1]	Enables switch-dependent halt when executing code from this segment
not_stack	[42: 1]	Stack indicator (0=stack, 1=not stack)
present_for_IO	[41: 1]	Segment-is-present indicator for use as an I/O buffer (0=absent, 1=present)
	[35: 4]	Reserved for software use
address	[31:32]	If present_to_proc and present_for_IO are both zero, the address field is interpreted by software; otherwise, it contains the base address of the actual segment

Actual Segment Descriptor, Second Word (ASD2)



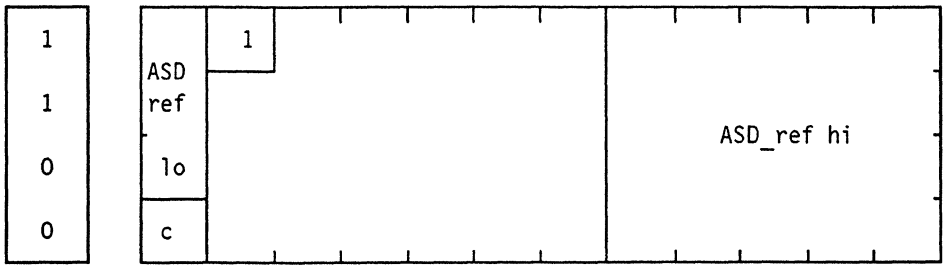
	[47: 8]	Reserved for software use
nybbles	[39: 4]	4-bit frames in addition to net words
deficit	[36:16]	Words less than AS_length-derived length
AS_length	[19:20]	Actual-segment length in words

Untouched CSD (Code Segment Descriptor)



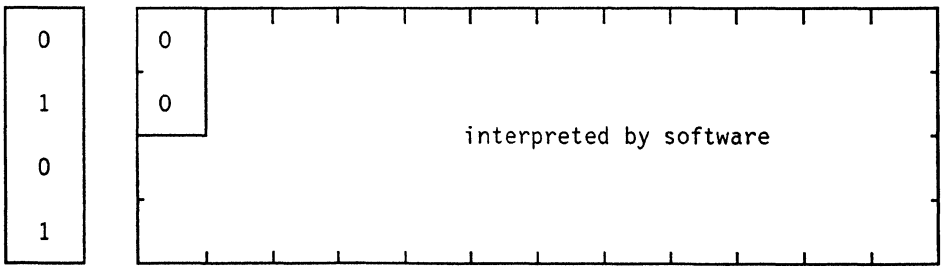
	[47: 2]	Binary 00
	[45:46]	Interpreted by software

Touched CSD



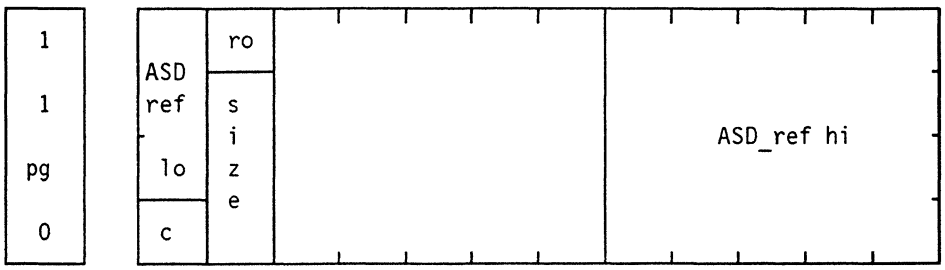
copy	[44: 1]	Copy bit (0=original, 1=copy) [0 by convention]
read_only	[43: 1]	Read-only bit (1=read-only)
	[42:23]	Reserved for software
ASD_ref	[19:23]	ASD_number

Untouched DD



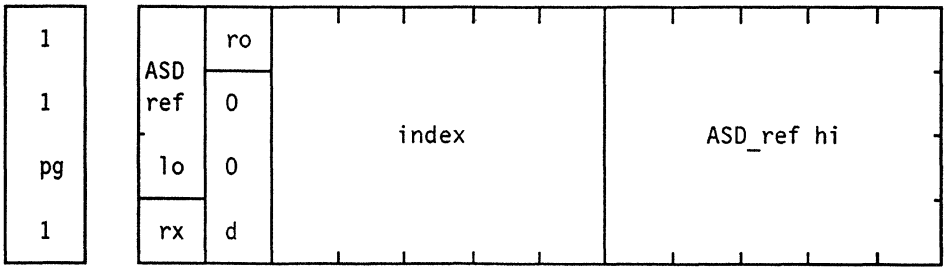
[47: 2]	Binary 00
[45:46]	Interpreted by software

Unindexed (Original or Copy) DD



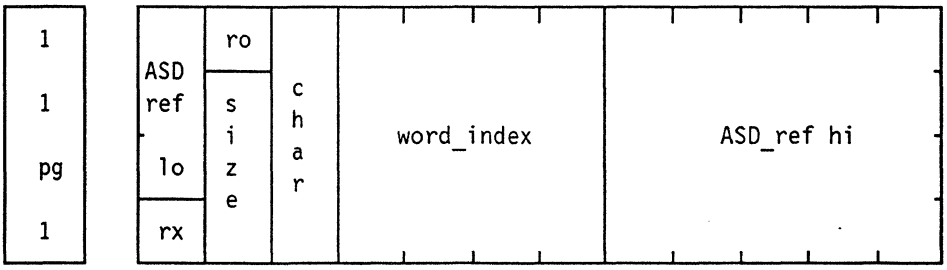
	tag	(C=unpaged, E=paged)
copy	[44: 1]	Copy bit (0=original, 1=copy)
read_only	[43: 1]	Read-only bit (0=read/write, 1=read only)
element_size	[42: 3]	The type of array element (0=single precision, 1=double precision, 2=hex, 4=EBCDIC; 3,5,6,7 are invalid)
	[39:20]	Reserved for software
ASD_ref	[19:23]	ASD_number

IndexedWordDD (Indexed Word Data Descriptor)



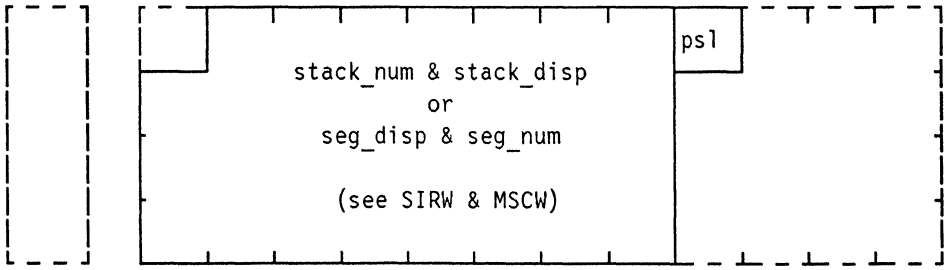
	tag	(D: unpaged, F: paged)
reindex	[44: 1]	Controls indexing, Data Array, and set-element_size operators (0=disabled, 1=enabled)
read_only	[43: 1]	Read-only bit (0=read/write, 1=read only)
element_size	[42: 3]	The type of array element (0=single precision; 1=double precision). (2,4 denote IndexedCharDD; 3,5,6,7, are invalid)
index	[39:20]	The word index from the base of the array to the referenced item
ASD_ref	[19:23]	ASD_number

IndexedCharDD (Indexed Character Data Descriptor)



	tag	(D: unpaged, F: paged)
reindex	[44: 1]	Controls indexing, Data Array, and set-element_size operators (0=disabled, 1=enabled)
read_only	[43: 1]	Read-only bit (0=read/write, 1=read only)
element_size	[42: 3]	The type of array element (2=hex, 4=EBCDIC). (0,1 denote IndexedWordDD; 3, 5, 6, 7 are invalid)
char_index	[39: 4]	The index within the word of the referenced character
word_index	[35:16]	The index from the base of the array to the word containing the referenced character
ASD_ref	[19:23]	ASD_number

Environment Link



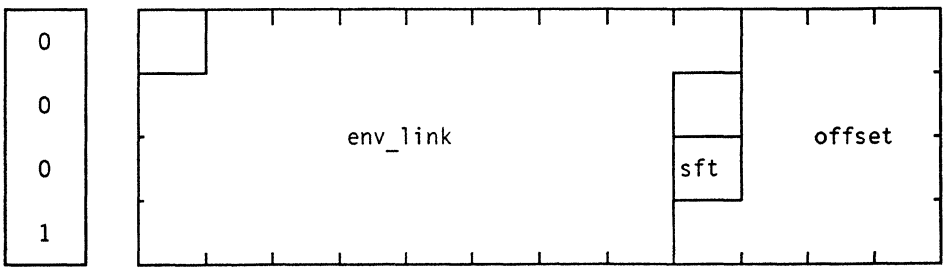
Normal Environment Link

stack_num	[46:15]	The stack_number identifying the stack containing the referenced location
stack_disp	[31:16]	The displacement from the base of the stack to the MSCW of the normal activation record
pseudolink	[15: 1]	0: denotes normal env_link

Pseudo Environment Link

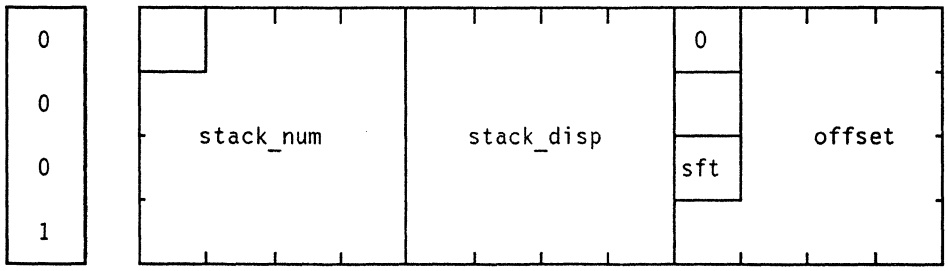
seg_num	[46:23]	The ASD_number identifying the segment containing the referenced location
seg_disp	[23: 8]	The displacement from the base of the segment to the MSCW of the pseudo activation record
pseudolink	[15: 1]	1: denotes pseudo env_link

Stuffed Indirect Reference Word (SIRW)

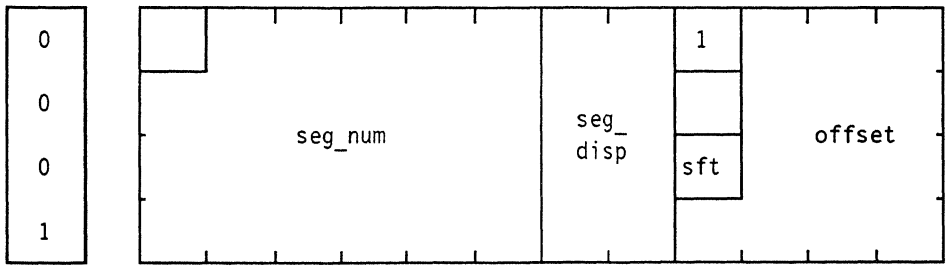


env_link	[46:32]	Designates activation record in which reference occurs
	[13: 1]	Reserved for software
offset	[12:13]	The offset from the base of the activation record to the referenced location

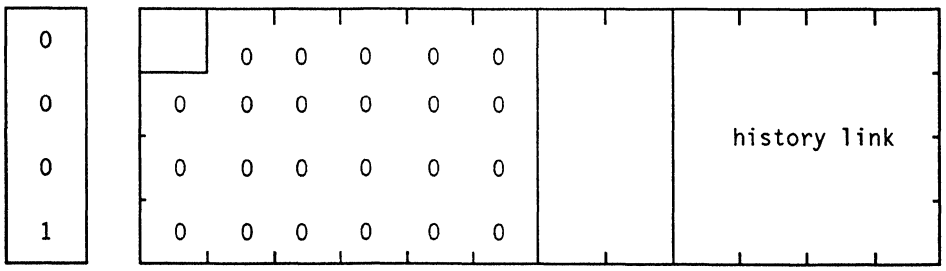
SIRW with normal env_link



SIRW with pseudo env_link

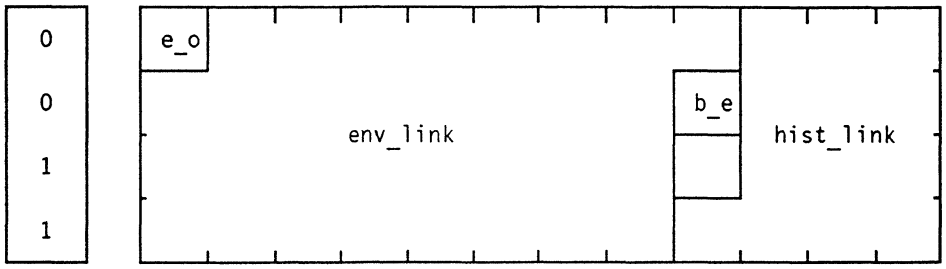


Inactive Mark Stack Control Word (MSCW)



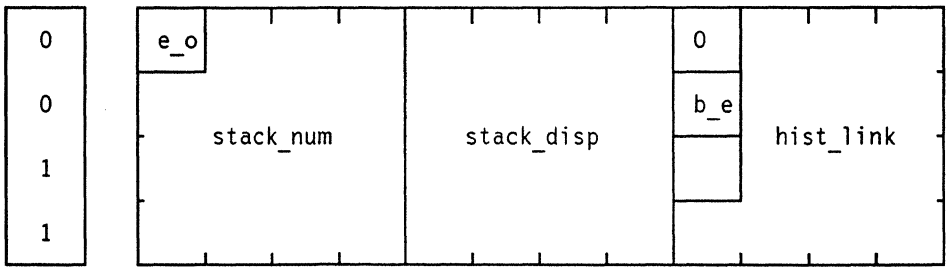
	[46:23]	Must be zero
history_link_I	[15:16]	Self-relative displacement to prior MSCW

Entered MSCW

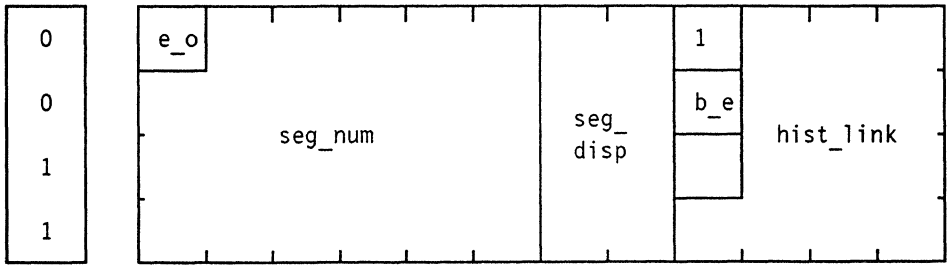


exit_opt	[47: 1]	0: no optimization information, 1: display update to D[old LL] for A16
env_link	[46:32]	Identifies the immediate global activation record (if lex_level greater than zero)
block_exit	[14: 1]	(0=disarmed, 1=armed)
history_link	[12:13]	Self-relative displacement to prior MSCW

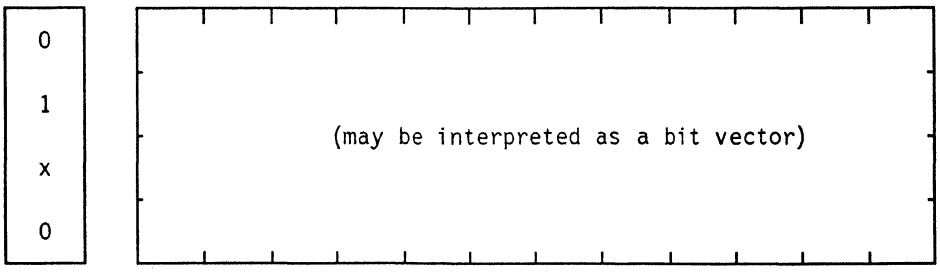
Entered MSCW with normal env_link



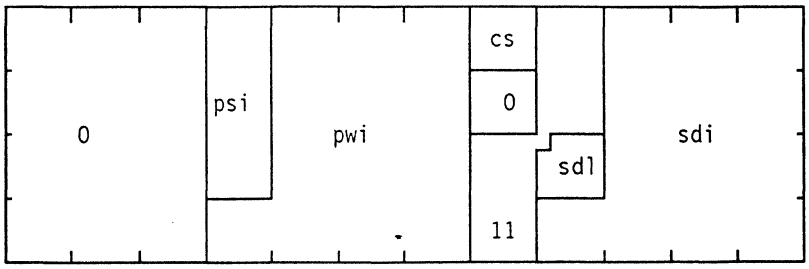
Entered MSCW with pseudo env_link



Tag-4 Word and Tag-6 Word

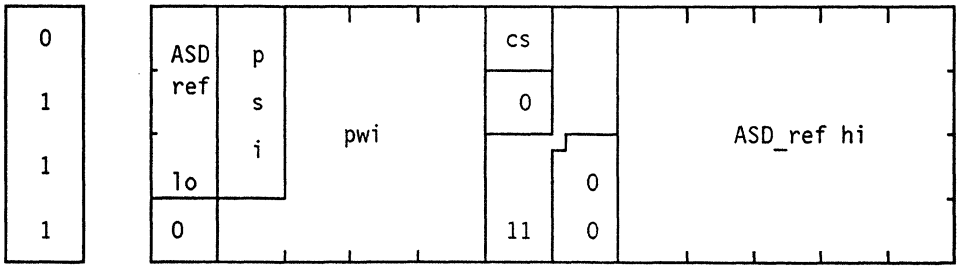


Program Control Word (PCW) Skeleton



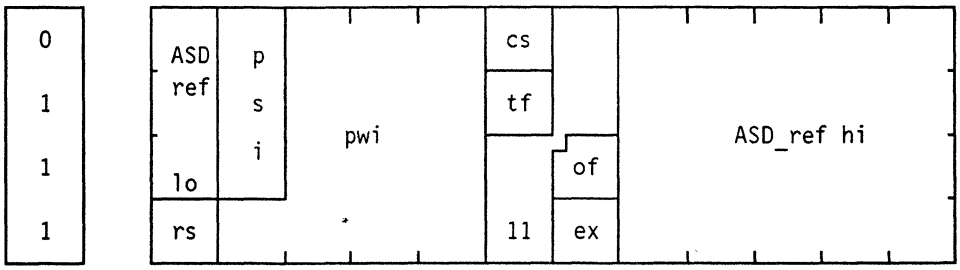
	[47:12]	Zero
skeleton_psi	[35: 3]	The PSI code stream pointer component
skeleton_pwi	[32:13]	The PWI code stream pointer component
skeleton_CS	[19: 1]	The initial value of the CS Boolean (0=normal state, 1=control state)
	[18: 1]	Must be 0 in user programs
skeleton_LL	[17: 4]	The lex level for the new activation record
sdll	[13: 1]	The lambda component of an address couple designating the CSD
sdi	[12:13]	The delta component of an address couple designating the CSD

Program Control Word (PCW)



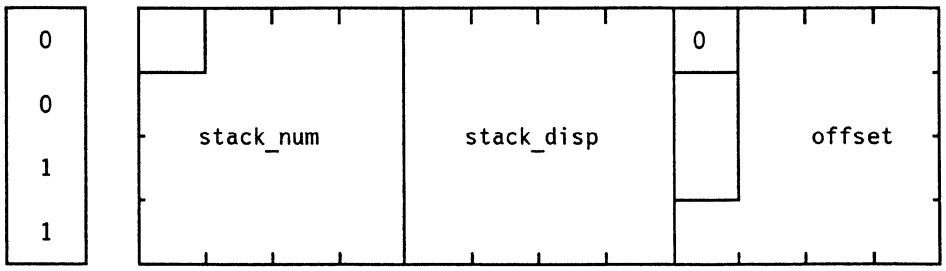
psi	[43: 3]	The PSI code stream pointer component
pwi	[40:13]	The PWI code stream pointer component
control_state	[27: 1]	CS Boolean (0=normal state, 1=control state)
	[26: 1]	Must be zero
lex_level	[25: 4]	The lex_level for the new activation record
ASD_ref	[19:23]	ASD_number

Return Control Word (RCW)



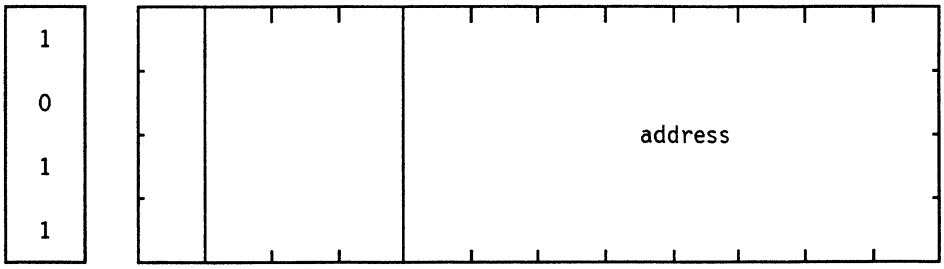
rs	[44: 1]	Restart indicator (0=initial, 1=restart mode)
psi	[43: 3]	The PSI code stream pointer component
pwi	[40:13]	The PWI code stream pointer component
control_state	[27: 1]	CS Boolean (0=normal state, 1=control state)
tfff	[26: 1]	TFFF (true false flip-flop)
lex_level	[25: 4]	The lex_level at which the activation record runs
offf	[21: 1]	OFFF (overflow flip-flop)
extf	[20: 1]	EXTF (external sign flip-flop)
ASD_ref	[19:23]	ASD_number

Top of Stack Control Word (TSCW)



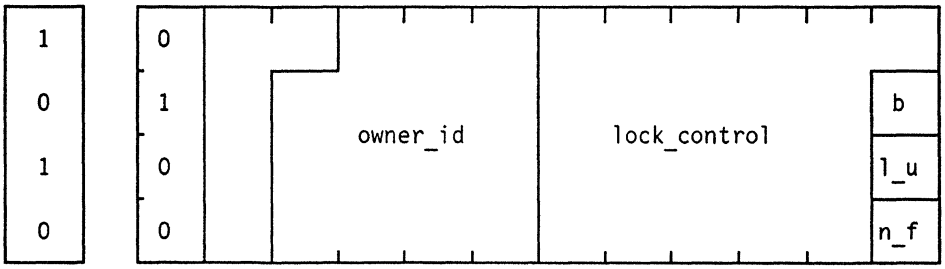
stack_num	[46:15]	The stack_number identifying the stack whose base contains this TSCW
stack_disp	[31:16]	Displacement to the topmost MSCW
	[15: 1]	Must be zero
offset	[12:13]	Offset of top of stack from topmost MSCW

Absolute Store Reference Word (ASRW)



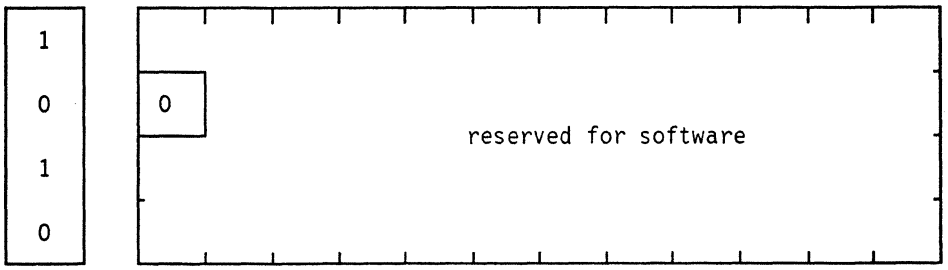
	[47: 4]	Reserved for coexistence
address	[31:32]	Address of memory item

Interlock Protected Object Word (POW) (Interlock after first use)



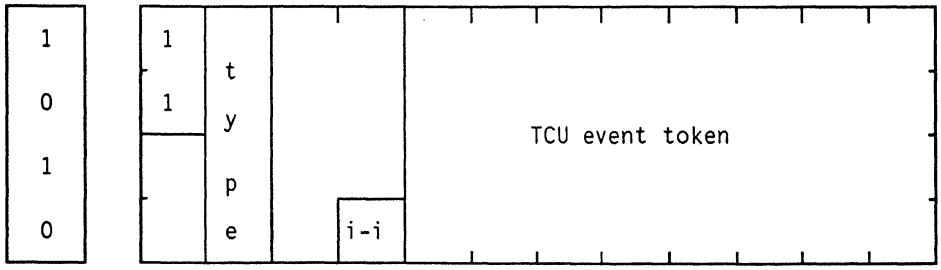
	[47: 4]	Binary 0100: denotes interlock POW
	[43: 5]	Reserved for software
owner_id	[38:15]	Stack number of lock owner
lock_control	[23:21]	Reserved for software use
broken_bit	[2: 1]	1 => state is Broken
lock_unc_bit	[1: 1]	1 => state is Lock_Uncontended
not_free_bit	[0: 1]	1 => state is not Free

Soft POW



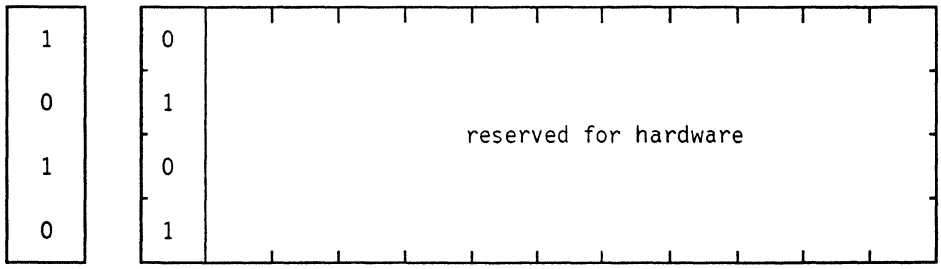
[46: 1]	0: denotes software POW
[45:47]	Reserved for software

Hard POW



	[47: 2]	Binary 11: denotes hardware POW
	[45: 2]	Reserved for software
type	[43: 4]	Binary 0011: denotes event
	[39: 7]	Reserved for software
i-i	[32: 1]	IOP inhibit: if 1, IOP will not cause event
TCU event token	[31:32]	TCU event token

Restart POW



[47: 4]	Binary 0101: denotes restart POW
[43:44]	Reserved for hardware

Appendix E
Restricted Operators

aMVST <Move to Stack>

The restricted operation aMVST changes the processor's site of activity by deactivating the current stack and activating a destination stack. The operation is defined as a common action invoked by action of the TCU (on A16 systems), which supplies the destination stack_number.

aMVST consists of the following functional steps:

1. **Deactivate the current stack:**

   Any valid top-of-stack registers are pushed onto the current stack.

   A new activation record is created on the current stack, as though a procedure were being entered. Two words are placed at the top of the stack: an entered MSCW and an RCW. The MSCW is created as though an interrupt or accidental entry were performed, with implicit MKST and ENTR operations; the Environment Link in the resulting MSCW is zero. The RCW is generated as defined for ENTR.

   A TSCW designating the new AR is placed at word 0 of the stack. TSCW.offset designates the RCW, even though the processor disregards that field for the destination stack.

2. **Establish the destination stack:**

   SNR is set from the destination stack_number. The stack_number is used to access the stack descriptor (an ASD). The word fetched from ASD1[stack_number] must be an ASD1 with not_stack = 0 and present_to_proc = 1. The word in ASD2 must be an ASD2; its AS_length defines the length of the stack for purposes of Stack Overflow detection. The base word of the stack must be a TSCW, which must designate an activation record at the top of the stack as created by an aMVST operation executed in the process associated with the destination stack when that process was last active.

   The processor consumes that topmost AR as though an exit operation was performed, establishing the state for resuming the process associated with the destination stack (its state prior to the aMVST operation performed in that process). This portion of the aMVST operation can be described as follows: F and S are set from the displacement and offset fields of the TSCW to designate the activation record at the top of the inactive destination stack. LL is set to an arbitrary value, D[LL] is set to F, and an Exit operation is performed. The Exit operation is executed as either an EXIT or RETN:

   a. EXIT if the TCU command specifies no result, which is the case if the destination stack was most recently deactivated by a TCU-initiated request's command.

b. RETN if the TCU command specifies a result, which is the case if the destination stack was most recently deactivated by a TCU acknowledge's command in response to message generated by the processor (caused by the restricted operator, XFNC).

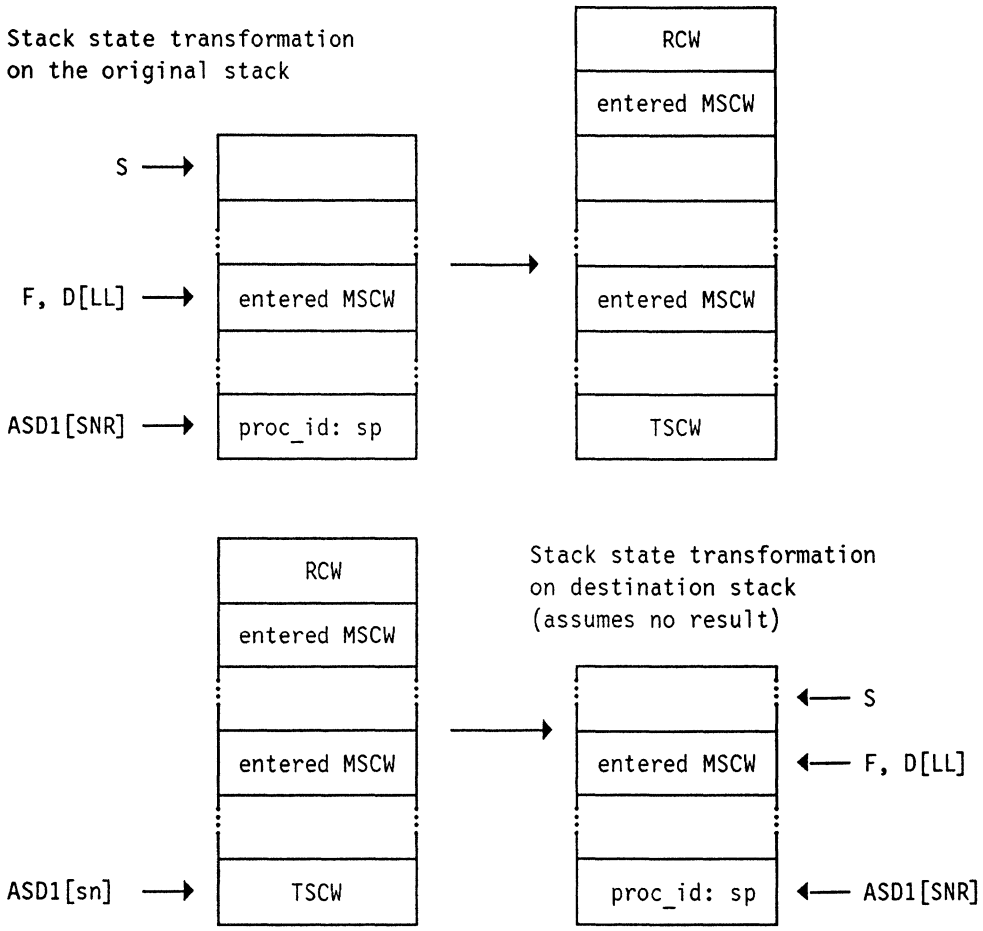
The proc_id value (as a single_integer) is stored at the base word of the destination stack.

It must be noted that grouping the previous two steps into a single operation is a documentation convenience. In fact, the TCU can cause the processor to deactivate the current stack without establishing a destination stack by sending a "Go Idle" command. Also, the TCU causes the processor to establish a destination stack without deactivating a current stack by sending a "Change Stack" command while the processor is in idle state.

Interrupt handling in aMVST is as follows:

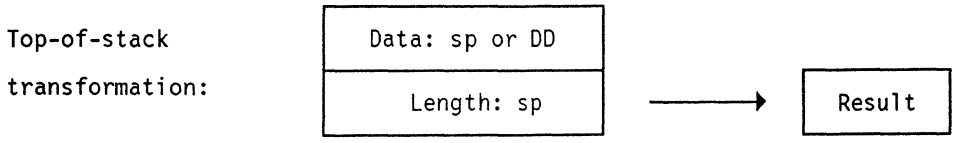
- Stack Overflow interrupt can occur as any top-of-stack registers are pushed onto the stack prior to the generation of the new AR in Step 1 of aMVST; in this case, the processor will be conditioned to repeat the aMVST operation upon exit from the interrupt procedure. Stack Overflow interrupt is disabled during the generation of the new AR and for the remainder of the aMVST operation; it is enabled at the conclusion of the operation.
- A frame overflow condition, which the user can cause, might be detected while generating the new AR in Step 1 of aMVST. The resulting Structure Error is not resumable, but the interrupt is reported normally (in the old stack). The processor never sends an acknowledge to the TCU, so the TCU knows that the stack was not deactivated.
- If any interrupt condition is detected from the time that generation of the new AR on the initial stack is finished until the topmost AR on the destination stack is consumed, the processor is not running on a valid stack. Therefore, instead of generating the interrupt, the processor immediately generates a superhalt condition. If a superhalt occurs, the Interrupt_Count value is undefined.
- A Service Request interrupt can be required by the "Change Stack" command. The interrupt is taken after the topmost AR on the destination stack is consumed and after the result, if any, is pushed.
- If an Absent Segment condition exists for the code segment designated by the RCW in the topmost AR, the normal service interrupt is generated, as for an Exit operation.

The following diagrams show the main effects of aMVST on the original and destination stacks. The F register is shown designating the same MSCW as D[LL], which will be the case unless there is an inactive MSCW in the topmost AR.



CSCP <Communicate with System Control Processor>

The restricted operator CSCP generates an Unimplemented Operator interrupt on the A16. It has two arguments and returns a result provided by the software interrupt routine.



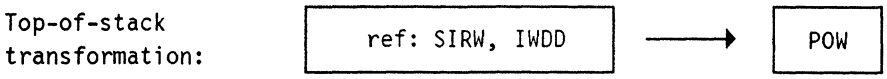
HALT <Conditional processor halt>

The restricted operator HALT is a conditional form of STOP. If the processor Halt Boolean is false, HALT is equivalent to NOOP. If the Halt Boolean is true, HALT is equivalent to STOP.

LPOW <Load POW>

LPOW is a restricted operator that performs a Read Evaluation of a reference chain to fetch a POW.

```
<Initial Reference>      ::=  {SIRW, IndexedWordDD}
<target>                 ::=  {operand, POW}

SIRW            ──────►  SIRW
                         IndexedWordDD
                         <target>

IndexedWordDD   ──────►  <target>
```



If the target is an operand, an Activate Object interrupt is generated. If it is a POW, an exact copy of the POW is left on the stack as the operator result. (The element_size of an IndexedWordDD is disregarded.)

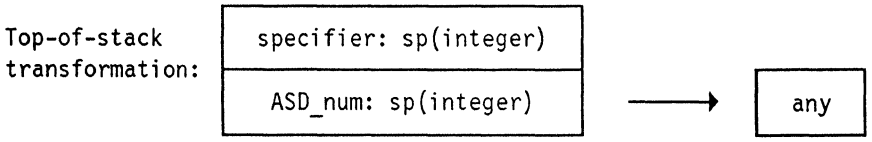
The reference chain for LPOW is not defined to include PCWs, but the A16 accepts them and performs accidental entry by invoking common action aACCE; the procedure result becomes a new initial reference.

RASD <Read ASD-table word>
RAST <RASD trapping synonym>

Normally RAST is undefined; however, a special maintenance mode available on A16 systems causes RASD and WASD to generate Unimplemented Operator interrupts and allows RAST and WAST to be used in their place.

The restricted operator RASD reads one word of the ASD table. RASD accepts a 23-bit integer argument as an ASD_number and a 2-bit integer argument as a word specifier. The word accessed is ASDi[ASD_number], where i-1 is the word specifier.

RASD performs a load-transparent operation, returning the contents of the designated ASD-table word.



REXI <Return/exit from interrupt>

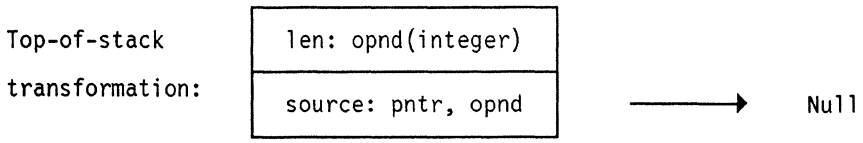
The restricted operator REXI is a synonym for EXIT on the A16.

RUNI <Indicate running>

The restricted operator RUNI performs no action.

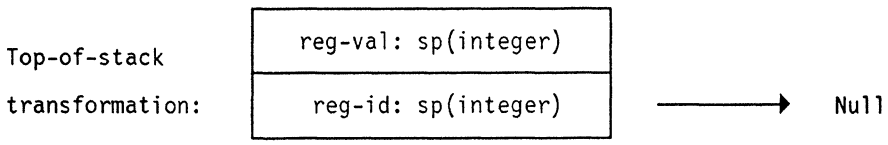
SHOW <Primitive display>

The restricted operator SHOW generates an Unimplemented Operator interrupt on the A16. SHOW requires two arguments and the software interrupt routine does not return a result.



SPRR <Set processor register>

The restricted operator SPRR assigns a value to a register; it requires two stack arguments. The second argument must be a 6-bit integer whose value is a valid register id. The top argument is the register value; it must be a k-bit integer, where k is the width of the destination register.



The contents of the specified register are set to the register value, except when the register id is 36, for which special action is defined later in this description. The only defined actual registers are D[LL] (whose id is LL, the current lex level), F (38), S (52), and SNR (53).

Register id values 36 and 37 designate LOSR and BOSR, respectively. RPRR reports 0 for register id 37. SPRR performs a special operation for register id 36: the register value argument is disregarded (but it nevertheless must be a 16-bit integer), and the Stack Overflow detection mechanism is enabled.

SPRR with register id 42 performs a special action: the processor purges its ASD cache. The register value is not significant, but must be a 16-bit integer.

The following table specifies the decimal register id encodings, register names, validity for RPRR and SPRR, and register widths (in bits). All register id values not listed are invalid.

register id	register name	RPRR	SPRR	width
LL	D[LL]	yes	invalid	16
36	(LOSR)	invalid	special	
37	(BOSR)	yes	invalid	0
38	F	yes	yes	16
42		invalid	special	
52	S	yes	yes	16
53	SNR	yes	invalid	15

If the register id designates S, the new register value remains in S at the completion of the SPRR operator (the arguments to SPRR are consumed before the register assignment is made).

The invariant S >= F must be preserved in any assignment of a value to S or F.

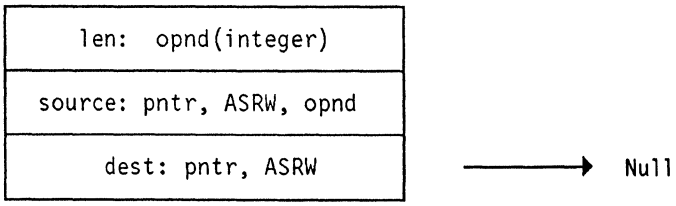
STOP <Unconditional processor halt>

The restricted operator STOP causes the processor execution to halt in an orderly way so that execution can be resumed by an external action. The stack state is unaffected.

TWOD <Transfer Words Overwrite Delete>

The restricted operator TWOD copies source words into a destination sequence without regard to tags. TWOD requires three arguments: a length, source, and destination.

Top-of-stack transformation:



The length must be an integer greater than or equal to zero, or the operation is undefined. If the length is zero, no transfer occurs.

The source must be an operand, an IndexedWordDD, an IndexedCharDD with char_index equal to zero, or an ASRW. The sequence defined by an operand is the source argument concatenated with itself indefinitely; if the source is a double precision operand and the second word is not identical to the first, the result is undefined.

The destination must be an IndexedWordDD, an IndexedCharDD with char_index equal to zero, or an ASRW. A source or destination IndexedDD or ASRW defines a sequence consisting of the designated word and its physical successors (the words with the next higher memory addresses). The TWOD operand is undefined if the source or destination argument is an IndexedDD and the sequence is not entirely within the designated actual segment; the following inequality must hold:

```
IndexedDD.index + length  <= ASD[IndexedDD.ASD_ref].AS_length
```

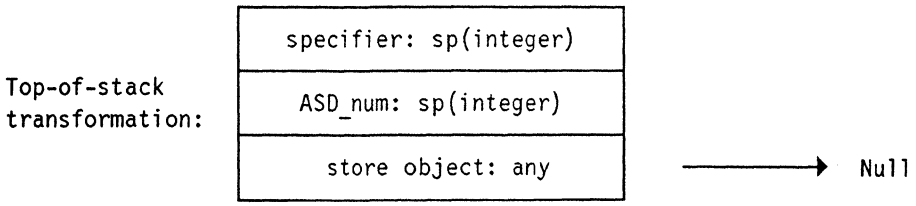
Each word of the source sequence,including its tag, is transferred to the destination sequence; prior destination contents are ignored. The operation continues until the number of words specified by length have been transferred.

WASD<Write ASD-table word>
WAST <WASD trapping synonym>

Normally WAST is undefined; however, a special maintenance mode available on A16 systems causes RASD and WASD to generate Unimplemented Operator interrupts and allows RAST and WAST to be used in their place.

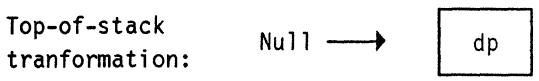
The restricted operator WASD writes one word of the ASD table. WASD accepts a 23-bit integer argument as an ASD_number and a 2-bit integer argument as a word specifier. The word accessed is ASDi[ASD_number], where i-1 is the word specifier.

WASD overwrites a store object of any type into the designated ASD-table word.



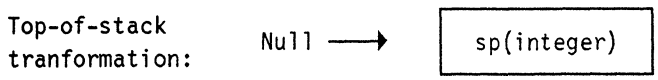
WATI <Read machine identification>

The restricted operator WATI leaves on top of the stack a double precision operand containing the proc state, except for the proc_id.



WHOI <Read proc_id>

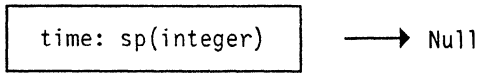
The restricted operator WHOI leaves on top of the stack a single_integer containing the processor identification number, proc_id.



WTOD <Write Time Of Day clock>

The restricted operator WTOD sets the time of day clock from an operand on top of the stack. The argument must be a 36-bit integer.

Top-of-stack tranformation:

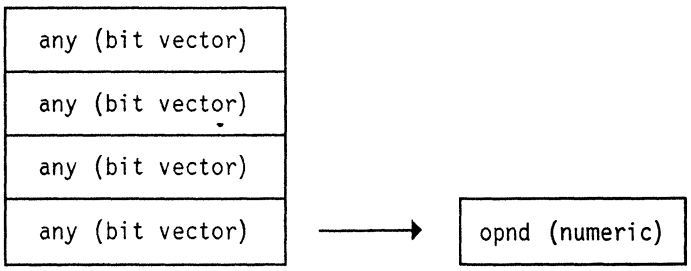


XFNC <system function call>

The restricted operator XFNC communicates with the external functional units, the TCU, IOU, and MIU. XFNC is only defined for the A16 and other systems requiring such communications.

It takes four words (items of any single-word type) as stack arguments and returns an operand as the result.

Stack state transformation:



In practice, the topmost bit vector is in the following fixed format, generated by system software:

```
[47:08]  =  0
[39:04]  =  Destination Module ID (MSM requestor port number)
[35:20]  =  0
[15:04]  =  Destination Unit ID  (IOU = 4'D',
                                  TCU = 4'E',
                                  MIU = 4'F'   IOM systems)
[11:08]  =  External Function ID
[03:04]  =  4'F' (Source Unit ID = CPM)
```

The four stack items are turned into a message that is sent to the requestor specified in the destination module ID field. The top word is the command word of the message and the next three words are commonly referred to as MSG_DATA_WORD_1 through MSG_DATA_WORD_3. When a message is issued, the CPM waits for an acknowledge message.

The acknowledge command may be one of two basic types:

1. **Continue:** the result from the acknowledge message is placed in TOS. If the acknowledge command requires an interrupt, an External Service interrupt is generated; otherwise, normal code execution resumes.
2. **Change Stack or Go Idle:** see aMVST. The current stack is deactivated without placing XFNC's result into the stack; that is done when this stack is established again.

Because XFNC always leaves a result, all external functions return a result. The CPM does not recognize "continue with no result" and returns whatever is in MSG_DATA_WORD_1 (joined to MSG_DATA_WORD_2 if the command indicates a double precision result) unconditionally if the acknowledge is a "continue". All functions that allow a change stack acknowledge require that a result be returned when the stack is established again; the CPM relies on the TCU to indicate whether a result is returned to the new stack by a "Change Stack" command.

Note: *Non-zero values for [47:08] and [35:20] in the topmost input bit vector are likely to cause either a Hardware Error Interrupt or system failure.*

ZIC <Zero Interrupt_Count>

The restricted operator ZIC sets the Interrupt_Count to zero. The stack is unaffected.

Program Restrictions Due to Hidden State

The concept of hidden state and some of the restrictions imposed on software are introduced in Section 4. Additional restrictions that are imposed on system programming are defined here.

The WASD operator is the only defined mechanism for changing the ASD table. A processor may capture the state of an ASD and hold it until that processor performs a WASD on that ASD or a move-stack operation.

In general, software must ensure that no processor could be accessing a segment before changing its ASD, except:

- An ASD marked absent can be changed to one marked present, providing that the last or only WASD performed to that ASD is the one that changes ASD1.present_to_proc from 0 to 1. (Note that ASD1.address can be changed in the same WASD.)
- ASD1.present_to_proc can be changed from 1 to 0, provided that the actual segment is neither a code segment (including table edit) nor a page table, does not contain any normal or pseudo activation record, and remains present at the same address; the software must take into account the fact that other processors may not be affected by the change until they perform a move-stack. (This action is permitted to allow software to determine experimentally whether a segment is in frequent use.) Note that this provisional resetting of present_to_proc does not establish the ASD as "absent" in the sense of the previous list entry; ASD1.present_to_proc can later be changed from 0 back to 1 without synchronization, but ASD1.address must remain unchanged.
- The processor can increase ASD2.AS_length for the current stack.

A processor may capture state from a page descriptor and its ASD between the time a copy of the descriptor is brought to the expression stack and the completion of an operation that consumes the descriptor. Software cannot safely change a page descriptor from touched to untouched, or alter the ASD3 for that page, if any process can be accessing that page.

Branch, enter, exit, and move-stack operators are the only mechanisms for altering the sequential execution of the code stream. The result is undefined if the program changes a code segment ASD or the contents of the code segment while any processor is executing code from the referenced segment. (The processor is free to capture the code-segment base address and limit, one or more words of code, and so forth.)

The effect is undefined of changing the environmental linkage for any activation record currently in the addressing environment of a processor.

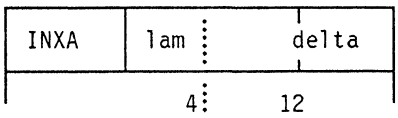
The result is undefined of any fetch or store operation in the current stack (whether via address couple, SIRW, DD, or absolute address) above the current upper bound for stack addressing. That bound is moved upward to the S setting by an explicit PUSH or the implicit push of an enter operator; it is moved downward by an exit operator or by a DLET operator that moves S below the most recent PUSH setting.

Appendix F
Operators To Be Phased Out

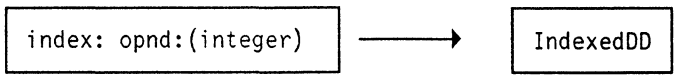
INXA <index via address couple parameter>

INXA is a three-syllable operator containing a fixed-fence address couple.

Operator format:



Top-of-stack transformation:

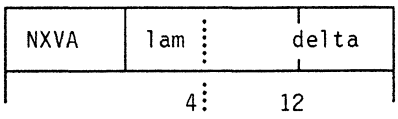


INXA is functionally equivalent to a name-call operator, containing the same address couple, followed immediately by the INDX operator.

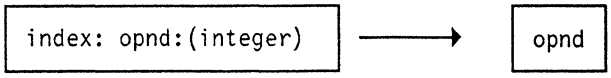
NXVA <index and load value via address couple parameter>

NXVA is a three-syllable operator containing a fixed-fence address couple.

Operator format:



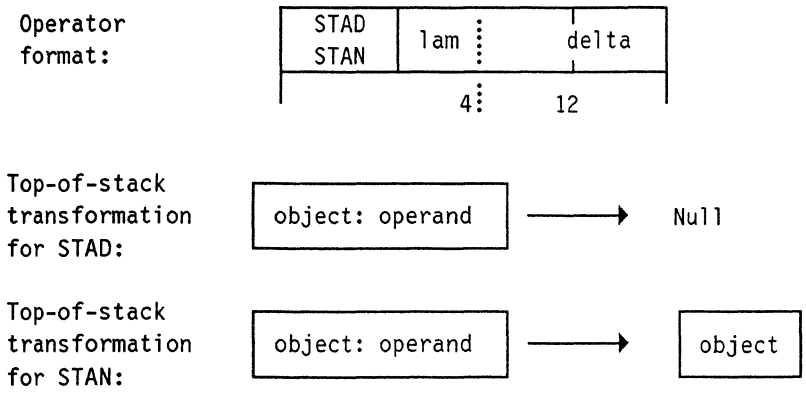
Top-of-stack transformation:



NXVA is functionally equivalent to a name-call operator, containing the same address couple, followed immediately by the NXLV operator.

STAD <store delete via address couple parameter>
STAN <store non-delete via address couple parameter>

STAD and STAN are three-syllable operators that contain a fixed-fence address couple; the code-stream appears as:



The STAD (or STAN) operator is functionally equivalent to a name-call operator, containing the same address couple, followed immediately by the STOD (or STON) operator, except that PCW evaluation requires the use of restart mode. Return from the accidental-entry function causes STAD or STAN to be resumed in restart mode; the operator ignores its code parameter and is functionally equivalent to STOD or STON, respectively.

Appendix G
Operator and Common Action Reference

This appendix lists each operator and common action, in alphabetical order by mnemonic. (Common actions have mnemonics beginning with lower case "a".) Several common actions are defined here for the purpose of consolidating the interrupt information for similar operations. Refer to "Common Actions" in Section 4 for more information.

If the operator is fully restricted, that fact is indicated after the operator name. The following information follows the name line:

p	primary operator
v	variant operator
e	edit mode operator (same encoding in table and single edit)
se	edit mode operator (single edit only)
te	edit mode operator (table edit only)

- The opcode is presented in hexadecimal notation, except for NAMC and VALC, which are in binary.
- Any code-stream parameters are defined following the opcode, separated by ampersands.
- If there are no ODI interrupts listed for this operator, a reference to another operator, or some other indication, appears at the right.
- Similar operators that generate the same interrupts are treated as a group, with one operator representing the rest of the group. Each represented operator refers to the representative, under which all the members of the group are named. Citations of the group take the form "DBUN etc." or "DBFL & DBTR" when there are only two in the group. An operator (like DBUN) "partly" represents others (like DBFL and DBTR) that can generate all the same interrupts plus more.

An instance of an operator-dependent interrupt is defined as a triple: an operator generates an interrupt for a reason. The interrupts and reasons are listed in this appendix under each operator and common action. All interrupts are summarized in Appendix H; the same instance triples are shown there by listing the operators/actions and reasons under each ODI.

A similar triple defines the relationship of a common action to its client operators: a client invokes a common action for a reason. The same triples are shown twice in this appendix: the clients and reasons are listed under each common action; the actions and reasons are listed under each operator.

For both interrupt instances and common action relationships, the form and content of the "reason" varies with the context. A typical interrupt entry shows the infraction and/or the data structure involved. A typical common action entry shows the purpose of the invocation and/or the "actual parameters": a common action is conceptually a "procedure" and most operate on "formal parameters".

The following notation is used in this appendix and Appendix H.

⟶	=	designates directly (via an ASD_number)
⟶⟶	=	designates indirectly (through an ASD or a display register)
[...]	=	indexed by ... (said of descriptors and the ASD table)
< >	=	not equal
|...|	=	absolute value
'		marks the update source/dest/edtab argument (internal or stack result)
ActSegLen(x):		actual segment length: ASD2.AS_length from ASD designated by x (default x is the object under discussion)
anticipation:		action performed before need
exhausted:		pointer designates last+1 element of virtual segment
Mem[x]:		the contents of the memory location at absolute address x
PageTable	=	unpaged unindexed SingleDD defining a page table: created from paged unindexed DD or from ASD3 referenced by paged indexed DD by effectively resetting tag and element_size
Stk[x]	=	Mem[ASD1[SNR].address+x]

Unless restart mode is specified explicitly, the interrupt descriptions that follow apply to the operator in initial mode.

This appendix and Appendix H define ODIs on the A16; the great majority of these cases are detected on all MCP/AS (Extended) systems.

aACCE <accidental entry>

This common action enters the procedure when a PCW is encountered in a reference chain. Refer to "Procedure Entry Operators" in Section 4 for more information on aACCE.

```
Clients:
   EVAL,
   STOD & STON:      SIRW ⟶▸ {PCW}
   VALC & LVLC:      parameter or SIRW ⟶▸ {PCW}
Interrupts:
   See aEVCH:        display update
   See aPRCW:        distribute entry PCW code-stream pointer
   Structure Error:  Stk[F].history_link_I >= 2**13 (frame overflow)
```

aAPIX <apply index to data descriptor>

This common action is a procedure with two formal parameters: a DD and an index value vI to be applied to the DD. The index value is a number of elements for the described array. Some clients pass I2DDs as I1DDs in order to reindex them in units of single words. Refer to "aAPIX <apply index to descriptor>" in Section 4 for more information.

The actual parameters for each invocation are shown as "DD by vI". For Data Array operators with an effective length of zero, vI is changed to zero and no interrupts are generated.

```
Clients:
   aCHAR:              client ICDD (as I1DD) by +1
   aFOP, aSOP:         client I2DD (as I1DD) by +1      (to access 2nd word)
   BMS:                current domain by -1
   CHEK:               current vector_A by stride_A
   CPV etc., CPVD,
   CPVS, DOT, DOTX:    current vector_A by stride_A
                       current vector_B by stride_B
   FMN, etc.:          current vector_A by stride_A
   GATH:               current vector_A by stride_A
                       vector_B argument by referent(vector_I)
                       current vector_I by +1
   INDX:               DD argument by RaI(index argument)
   NXLN:               SingleDD argument by RaI(index argument)
   NXLV:               WordDD argument by RaI(index argument)
   POLY:               current vector_A by stride_A
                       current vector_B by stride_B
```

```
   SCAT:              vector_A argument by referent(vector_I)
                      current vector_B by stride_B
                      current vector_I by +1
   SEQ:               current vector_A by stride_A
                      current vector_B by stride_B
   SE8C & SE4C:       argument pointer by 0 (after element_size change)
   SFDC:              current dest ICDD by +max(length,0)
   SFSC:              current source ICDD by +max(length,0)
   SRCH:              current domain by -1
   SRDC:              current dest ICDD by -max(length,0)
   SRSC:              current source ICDD by -max(length,0)
   SUM & SUMA:        current vector_A by stride_A
   SWFD etc.:         set by WordIndex(c)
   TRNS:              trtab by WordIndex(c)
   TWFD etc.:         set by WordIndex(c)
   TWOD:              current source or dest pointer (as I1DD) by +1
   TWSD & TWSU:       current source or dest pointer (as I1DD) by +1
   VMV, VMNV, VOV, VPS, VPV, VPVS, VSPV,
   VTS, VTV, VUV:     current vector_A by stride_A
                      current vector_B by stride_B

Interrupts:

   Absent Data:       client paged unindexed DD (page table)
                      ASD3 for client paged indexed DD (page table)
   Bounds Error:      The following conditions detected by aAPIX cause
                          client to generate interrupt if resulting
                          (unrepresentable) DD is used or returned:
                             new virtual index < 0
                             unpaged DD and new index >= 2**20
                             unpaged CharDD and new word index >= 2**16
                             new page index > ActSegLen(PageTable)
                             new page index = ActSegLen(PageTable)
                                  and new word index + new char index > 0
   Structure Error:   ASD3 for paged indexed DD is not unpaged indexed DD
                         (not checked if page index already known)
                      ASD3 ⟶ not ASD1/ASD2 (page table)
                      paged unindexed DD ⟶ not ASD1/ASD2 (page table)
                      PageTable[new page index]
                         not {unpaged unindexed DD, untouched DD}
   Touch Reference:   PageTable[new page index] is untouched DD
```

For convenience, Bounds Error conditions arising from attempts to generate unrepresentable indexed DDs are shown under aAPIX. However, not every instance can occur for every client: some pass only IWDDs or ICDDs; some can only increase or only decrease the virtual index. Furthermore, errors cannot be reported unless the indexed DD is evaluated or returned, so the actual interrupt generation is the responsibility of the client. For example, non-edit array operators that index a pointer or vector in anticipation of another iteration cannot generate an interrupt unless the anticipated iteration (or pointer update) occurs.

aCHAR <access word via ICDD>

This interrupt-consolidation common action accesses the designated word via an ICDD and advances the ICDD to designate the successor word.

The clients are all the non-edit character operators and the edit operators that move data. Note that these operators all coerce IWDD pointers to ICDDs. If the effective length is zero, the aAPIX index is zero and no interrupts are generated.

```
Clients:

    CEQD etc.:          access word via source1 or source2 ICDD
    ICUD etc.:          access word via source ICDD
    INSU etc.:          access word via dest ICDD
    MCHR etc.:          access word via source or dest ICDD
    PKUD etc., SEQD etc.,
    SISO, SWFD etc.:    access word via source ICDD
    TEQD etc., TRNS, TUND & TUNU,
    TWFD etc.:          access word via source or dest ICDD
    UPUD etc.:          access word via dest ICDD

Interrupts:

    See aAPIX:          client ICDD (as I1DD) by +1
    See aEIXD:          evaluate current client ICDD

    Inv Target:         client dest ICDD ⟶▸ not operand
                        client source ICDD ⟶▸ not operand
```

aEAC <evaluate address couple>

This common action performs basic evaluation of an address couple.

```
Clients:

    aFOP:               evaluate parameter for client VALC
    MPCW:               evaluate (sdll, sdi) from parameter
    NAMC & LNMC:        evaluate parameter

Interrupts:

    Inv Arg/Par:        lambda > LL
```

aEIRW <evaluate SIRW>

This common action performs basic evaluation of an SIRW.

```
Clients:

    aFOP, aILRE:          evaluate client SIRW
    aINTE:                evaluate SIRW at F+1 or successor SIRW
    aSOP:                 evaluate client SIRW
    DBUN etc.:            evaluate SIRW
    ENTR:                 evaluate SIRW at F+1 or successor SIRW
    EVAL, INDX, LODT, LPOW, NXLN, NXLV, OVRD etc.,
    STOD & STON:          evaluate SIRW

Interrupts:

    See aEVLK:            SIRW environment_link evaluation

    Bounds Error:         displacement+offset >= ActSegLen
```

aEIXD <evaluate indexed DD>

This interrupt-consolidation common action performs basic evaluation of an indexed DD. If the DD is paged and the initial word index equals 2**13, change word index to zero and use page index equal to ASD3.index+1.

```
Clients:

    aCHAR:                evaluate current client ICDD
    aFOP:                 evaluate client IWDD
    aILRE:                evaluate client I1DD
    aSOP:                 evaluate client IWDD
    BMS:                  evaluate current domain
    LODT, LPOW:           evaluate IWDD
    NXLN:                 evaluate final indexed Single DD
    OVRD etc.:            evaluate I1DD
    SRCH:                 evaluate current domain
    SWFD etc.:            evaluate set[WordIndex(c)]
    TRNS:                 evaluate trtab[WordIndex(c)]
    TWFD etc.:            evaluate set[WordIndex(c)]
    TWOD:                 evaluate current source or dest pointer
    TWSD & TWSU:          evaluate current source or dest pointer

Interrupts:

    Absent Data:          client indexed DD
                          ASD3 for client paged index DD (page table)
    Bounds Error:         word index >= ActSegLen
                              (except when ultimate client can generate
                               Source or Destination Boundary interrupt)
                          page index >= ActSegLen(PageTable)
```

```
Structure Error:    client indexed DD ⟶ not ASD1/ASD2
                    ASD3 for paged indexed DD is not unpaged indexed DD
                         (not checked if page index already known)
                    ASD3 ⟶ not ASD1/ASD2 (page table)
                    PageTable[page index] not {unpaged unindexed DD,
                                               untouched DD}
Touch Reference:    PageTable[page index] is untouched DD
```

aEVCH <traverse environmental chain>

This common action consolidates the interrupts generated in traversing the environmental chain.

```
Clients:

    aACCE, aINTE,
    ENTR, EXIT etc.:    display update

Interrupts:

    See aEVLK:          MSCW environment-link evaluation

    Structure Error:    for each AR traversed on environmental chain:
                          MSCW.env_link or D register ⟶⟶ not entered MSCW
```

aEVLK <evaluate environmental link>

This common action effects the basic evaluation of an Environmental Link.

```
Clients:

    aEIRW:              SIRW environment-link evaluation
    aEVCH:              MSCW environment-link evaluation

Interrupts:

    Absent Data:        if pseudolink = 0:
                           ASD1[stack_num].present_to_proc = 0
                        if pseudolink = 1:
                           ASD1[seg_num].present_to_proc = 0
    Bounds Errors:      displacement >= ActSegLen
    Structure Error:    if pseudolink = 0:
                           stack_num ⟶ not ASD1/ASD2
                        if pseudolink = 1:
                           seg_num ⟶ not ASD1/ASD2
```

aFOP <fetch operand value>

This common action fetches an operand value via an SIRW, address-couple parameter, or IWDD. Refer to "Read Evaluation Operators" in Section 4 for more information on aFOP. Validity checking of the target is performed by aFOP except when a non-operand value must be examined by the client as a possible reference or valid target; aFOP returns any non-operand argument to the client when fetching:

- for VALC via address-couple parameter or SIRW
- for LOAD via SIRW or I1DD

For vector operators with a zero length, aFOP does nothing.

```
Clients:

    CHEK:              fetch via current vector_A
    CPV etc.,
    CPVD, CPVS:        fetch via current vector_B
    DOT, DOTX:         fetch via current vector_A or vector_B
    FMN etc.:          fetch via current vector_A
    GATH:              fetch via current vector_B or vector_I
    LOAD:              fetch via SIRW or IWDD
    NXLV:              fetch via final indexed WordDD
    POLY:              fetch via current vector_A or vector_B
    SCAT:              fetch via current vector_B or vector_I
    SEQ:               fetch via current vector_B
    SUM & SUMA:        fetch via current vector_A
    VALC & LVLC:       fetch via parameter or subsequent reference
    VMV, VMVN, VOV:    fetch via current vector_A or vector_B
    VPS:               fetch via current vector_B
    VPV, VPVS, VSPV:   fetch via current vector_A or vector_B
    VTS:               fetch via current vector_B
    VTV, VUV:          fetch via current vector_A or vector_B

Interrupts:

    See aAPIX:         client I2DD (as I1DD) by +1 (to access 2nd word)
    See aEAC:          evaluate parameter for client VALC
    See aEIRW:         evaluate client SIRW
    See aEIXD:         evaluate client IWDD

    Inv Target:        2nd word via SIRW or address couple not operand
                       I1DD ⟶⏵ not operand (except for LOAD)
                       I2DD ⟶⏵ not operand
                       2nd word via I2DD not operand
```

aILRE <interlock reference evaluation>

This interrupt-consolidation common action performs the read and store evaluations for the interlock operators.

```
Clients:
    LOK, UNLK:          interlock reference evaluations

Interrupts:
    See aEIRW:          evaluate client SIRW
    See aEIXD:          evaluate client I1DD

    Inv Access:         read_only I1DD
    Inv Arg/Par:        interlock reference not {SIRW, I1DD, ASRW}
    Inv Target:         reference ⟶▸ not {sp operand, POW}
    Uninit Arg:         interlock-reference tag in {6, 4}
```

aINTE <interrupt entry>

This common action generates an interrupt, which is implemented as entry to an operating-system procedure designated by a processor-generated reference. Thereference is an SIRW whose env_link and offset fields are set according to the interrupt situation.

The action can be defined as four steps:

1. Invoke MKST
2. Place the interrupt reference on the stack (at Mem[F+1])
3. Place any parameter words on the stack
4. Invoke ENTR

```
Clients (not listed):   all operations that generate interrupts

Interrupts:
    See aEIRW:          evaluate SIRW at F+1 or successor SIRW
    See aEVCH:          display update
    See aPRCW:          distribute entry PCW code-stream pointer

    Missing PCW:        SIRW ⟶▸ not {SIRW, PCW}
    Structure Error:    Stk[F].history_link_I >= 2**13 (frame overflow)
```

aMVST <move to stack>

The MVST operation includes several parts:

An implicit MKST on the current stack generates no interrupts.

An implicit pseudo-ENTR on the current stack can interrupt due to frame overflow. ALL other interrupts of ENTR are prevented by the close coupling of the ENTR and MKST operations and by the fact that there is no reference, PCW, or code segment, and no display update.

Building and storing the TSCW on the current stack generates no interrupts.

Several interrupts can occur as the operator attempts to locate the new stack and fetch its TSCW. Distribution of the TSCW generates none.

Finally, an implicit EXIT on the destination stack is subject to all of the interrupts of that operator.

Interrupts same as EXIT, plus:

```
Stucture Error: Stk[F].history_link_I >= 2**13 after implicit MKST
                stack-number ——→ not ASD1/ASD2
                destination Stk[0] not TSCW
```

aPOAV <pointer operator argument validation>

This common action consolidates the interrupts generated during the validation of the arguments to pointer operators. The action can be described as a procedure with two to four formal parameters, each of which names an argument to be validated. The arguments for each client operator are listed in order up the stack (TOS last). The valid type or set of types for each argument (except length) is appended to the name. For length, the upper limit is specified.

The clients are all non-edit pointer operators and the enter-edit operators, which validate arguments for edit operators. The term "pointer" is interpreted only here to include unpaged unindexed copy DDs.

```
Clients:

    CEQD etc.:          validate source1:pointer, source2:{pointer,opnd},
                                length:MaxInx
    EXPU:               validate dest:pointer, length:MaxInx
    EXSD & EXSU:        validate dest:pointer, source:{pointer, opnd},
                                length:MaxInx
    ICUD etc.:          validate source:{pointer, opnd}, length:23
    PKUD etc.:          validate source:{pointer, opnd}, length:24
    SEQD etc.:          validate source:{pointer, opnd},
                                length:MaxInx, delim:sp operand
    SISO:               validate source:{pointer,opnd},
                                length:{if source is hex then 24 else 12)
    SWFD etc.:          validate source:{pointer,opnd},
                                length:MaxInx, set:I1DD
    TEED & TEEU:        validate dest:pointer, source:{pointer, opnd},
                                edtab:indexed DD
    TEQD etc.:          validate dest:pointer, source:{pointer, opnd},
                                length:MaxInx, delim:sp operand
    TRNS:               validate dest:pointer, source:{pointer, opnd},
                                length:MaxInx, trtab:I1DD
    TUND & TUNU:        validate dest:pointer, source:{pointer, opnd},
                                length:MaxInx
    TWFD etc.:          validate dest:pointer, source:{pointer, opnd},
                                length:MaxInx, set:I1DD
    TWOD:               validate dest:{pointer,ASRW},
                                source:{pointer,ASRW,opnd}
                                length:MaxInx
    TWSD & TWSU:        validate dest:pointer, source:{pointer, opnd},
                                length:MaxInx
    UPUD etc.:          validate dest:pointer, source:operand, length:24

Interrupts:

    Bounds Error:        for client source or destination pointer
                           (blocked by zero-length operator that deletes
                            its arguments)
                           IWDD.index > ActSegLen
                           ICDD.word_index > ActSegLen
                           ICDD.word_index = ActSegLen and .char_index > 0
                           IWDD.index >= 2**16
                               and client is not Transfer Words
```

```
Inv Arg/Par:        client length not operand or |RaI(length)| > limit
                    client argument not specified type
                    reindex disabled in client indexed DD
                         (source, dest, set, trtab, or edtab)
                    EBCDIC ICDD.char_index > 5
                    hex    ICDD.char_index > 11
Structure Error:    ASD3 for client paged indexed DD is not unpaged
                         indexed DD (not checked if page index known)
Uninit Arg:         client argument tag in {6, 4}
```

aPRCW <distribute PCW/RCW code stream pointer>

The common action aPRCW distributes a code stream pointer from a PCW or RCW, as follows:

1. PSN is set from the ASD_ref field of the PCW or RCW.
2. PWI and PSI are set from the pwi and psi fields, respectively. PWI must be in the range {0 to ASD2.AS_length-1} for the ASD referenced by the new PSN value; PSI must be in the range {0 to 5}.
3. If ASD1.present_to_proc is not set in the ASD designated by the new PSN value, an Absent Code interrupt is generated.

```
Clients:
    aACCE, aINTE:       distribute entry PCW code-stream pointer
    DBUN etc.:          distribute branch-dest PCW code-stream pointer
    ENTR:               distribute entry PCW code-stream pointer
    EXIT etc.:          distribute RCW code-stream pointer

Interrupts:

    Absent Code:        PCW or RCW
    Bounds Error:       (PCW or RCW).pwi >= ActSegLen
                        (PCW or RCW).psi > 5
    Structure Error:    PCW or RCW ⟶ not ASD1/ASD2
```

aSOP <store operand value>

This common action stores an operand value for operators that perform store evaluation of references. Refer to "Normal Store Operators" in Section 4 for more information on aSOP. aSOP accepts two formal parameters: the value to be stored and the reference. When the reference is an SIRW, the client (normal store) operator verifies that the target is a Data Word.

For vector operators with a zero length, aSOP does nothing.

```
Clients:

    CPV etc., CPVD, CPVS,
    GATH, SCAT, SEQ:      store via current vector_A
    STOD & STON:          store via SIRW or IWDD
    VMV, VMVN, VOV, VPS, VPV, VPVS, VSPV,
    VTS, VTV, VUV:        store via current vector_A

Interrupts:

    See aAPIX:            client I2DD (as I1DD) by +1  (to access 2nd word)
    See aEIRW:            evaluate client SIRW
    See aEIXD:            evaluate client IWDD

    Inv Access:           read_only IWDD
    Inv Arg/Par:          object is sp and reference is I2DD
                          object is sp and SIRW ⟶▸ dp opnd
                          object is dp and reference is I1DD
                          object is dp and SIRW ⟶▸ {sp opnd, tag-6 word,
                                                    tag-4 word}
    Inv Target:           IWDD ⟶▸ not Data Word
                          2nd word not Data Word
```

aVOAV <vector-operator argument validation>

This common action consolidates the interrupts generated during the validation of the arguments to vector operators. The action can be described as a procedure with four to six formal parameters, each of which names an argument to be validated. The arguments for each client operator is listed in order up the stack (TOS last). The valid type or set of types is appended to each name, except for length and stride arguments, which have fixed type.

Additional type checking is defined under several vector operators to enforce consistency of precision between arguments. Many of these checks are in anticipation of the type checking in aSOP.

```
Clients:

    CHEK:                 validate vector_A:I1DD, length, stride_A,
                                   hash:sp operand
```

```
    CPV etc.:          validate vector_A:IWDD, vector_B:{IWDD,opnd},
                               length, stride_A, stride_B
    CPVD:              validate vector_A:I2DD, vector_B:IWDD,
                               length, stride_A, stride_B
    CPVS:              validate vector_A:I1DD, vector_B:IWDD,
                               length, stride_A, stride_B
    DOT:               validate vector_A:I1DD, vector_B:I1DD, length,
                               stride_A, stride_B, sum:sp operand
    DOTX:              validate vector_A:IWDD, vector_B:IWDD, length,
                               stride_A, stride_B, sum:dp operand
    FMN etc.:          validate vector_A:IWDD, length, stride_A,
                               X:sp operand, V:operand
    GATH:              validate vector_A:IWDD, vector_B:IWDD,
                               length, stride_A, vector_I:I1DD
    POLY:              validate vector_A:IWDD, vector_B:{IWDD, opnd},
                               length, stride_A, stride_B, Z:operand
    SCAT:              validate vector_A:IWDD, vector_B:{IWDD, opnd},
                               length, vector_I:I1DD, stride_B
    SEQ:               validate vector_A:IWDD, vector_B:{IWDD,opnd},
                               length, stride_A, stride_B, Z:operand
    SUM & SUMA:        validate vector_A:IWDD, length, stride_A,
                               sum:operand
    VMV, VMVN, VOV:    validate vector_A:IWDD, vector_B:{IWDD,opnd},
                               length, stride_A, stride_B
    VPS:               validate vector_A:IWDD, vector_B:IWDD,
                               length, stride_A, stride_B, S:operand
    VPV:               validate vector_A:IWDD, vector_B:{IWDD,opnd},
                               length, stride_A, stride_B
    VPVS:              validate vector_A:IWDD, vector_B:IWDD,
                               length, stride_A, stride_B, S:operand
    VSPV:              validate vector_A:IWDD, vector_B:{IWDD,opnd},
                               length, stride_A, stride_B, S:operand
    VTS:               validate vector_A:IWDD, vector_B:IWDD,
                               length, stride_A, stride_B, S:operand
    VTV, VUV:          validate vector_A:IWDD, vector_B:{IWDD,opnd},
                               length, stride_A, stride_B
Interrupts:
    Bounds Error:      MaxInx + 1 > |stride| > MaxInx/3
    Inv Arg/Par:       client length not sp integer or |length| > MaxInx
                       client stride not sp integer
                            or |stride| > MaxInx
                       (client stride)*(restart count) > MaxInx
                            (restart mode)
                       client argument not specified type
                       client vector IWDD reindex disabled
    Structure Error:   ASD3 for client paged indexed DD is not unpaged
                            indexed DD (not checked if page index known)
    Uninit Arg:        client argument tag in {6, 4}
```

ADD <add>

p 80

```
Arith Overflow:   R(x+y) = TooBig                    [Exponent Overflow]
Arith Underflow:  N(R(x+y)) = TooSmall                          [spurious]
Inv Arg/Par:      x (TOS2) not operand or y (TOS) not operand
Uninit Arg:       TOS or TOS2 tag in {6, 4}
```

AMAX <arithmetic maximum>

v 958A
also represents AMIN

```
Inv Arg/Par:      x (TOS2) not operand or y (TOS) not operand
Uninit Arg:       TOS or TOS2 tag in {6, 4}
```

AMIN <arithmetic minimum>

v 9588 — See AMAX

ASRT <assert>

v 9580 & interrupt_code:8 — See AVER

AVER <aver>

v 958F
also represent ASRT

```
False Assertion:  Boolean (TOS) is False
```

BCD <binary convert to decimal>

v 9577 & N:8

```
Int Overflow:     RId(B) = TooBig
Int Arg/Par:      B (TOS) not operand
                  N > 24
Uninit Arg:       TOS tag in {6, 4}
```

BMS <bounded mask search for equal>

```
v   95A9

See aAPIX:          current domain by -1
See aEIXD:          evaluate current domain, if length > 0

Inv Arg/Par:        length (TOS) not operand or |RaI(length)| > MaxInx
                    domain (TOS2) not I1DD
                    domain reindex disabled
Structure Error:    ASD3 for paged indexed domain is not unpaged indexed
                        DD (not checked if page index known)
Uninit Arg:         length or domain tag in {6, 4}
```

BRFL <branch false>

```
p   A0 & op_psi:3 & op_pwi:13                                   See BRUN
```

BRST <bit reset>

```
p   9E & Db:8                                                   No interrupts
```

BRTR <branch true>

```
p   A1 & op_psi:3 & op_pwi:13                                   See BRUN
```

BRUN <branch unconditional>

```
p   A2 & op_psi:3 & op_pwi:13
also represents BRFL & BRTR

Bounds Error:       op_pwi >= ActSegLen(PSN)
                    op_psi > 5
```

BSET <bit set>

```
p   96 & Db:8                                                   No interrupts
```

CBON <count binary ones>

```
v   95BB

Inv Arg/Par:        TOS not operand
Uninit Arg:         TOS tag in {6, 4}
```

CEQD <compare characters equal delete>

```
p   F4
also represent CEQU, CGED, CGEU, CGTD, CGTU, CLED,
               CLEU, CLSD, CLSU, CNED, CNEU

See aCHAR:        access word via source1 or source2 ICDD
See aPOAV:        validate source1:pointer, source2:{pointer, opnd},
                           length:MaxInx
Inv Arg/Par:      source1 is EBCDIC and source2 is hex or vice versa
```

CEQU <compare characters equal update>

```
p   FC                                                    See CEQD
```

CGED <compare characters greater or equal delete>

```
p   F1                                                    See CEQD
```

CGEU <compare characters greater or equal update>

```
p   F9                                                    See CEQD
```

CGTD <compare characters greater delete>

```
p   F2                                                    See CEQD
```

CGTU <compare characters greater update>

```
p   FA                                                    See CEQD
```

CHEK <compute check-hash>

```
v   95C8

See aAPIX:        current vector_A by stride_A
See aFOP:         fetch via current vector_A
See aVOAV:        validate vector_A:I1DD, length, stride_A,
                           hash:sp operand
```

CHSN <change sign>

p 8E

Inv Arg/Par: TOS not operand
Uninit Arg: TOS tag in {6, 4}

CLED <compare characters less or equal delete>

p F3 See CEQD

CLEU <compare characters less or equal update>

p FB See CEQD

CLSD <compare characters less delete>

p F0 See CEQD

CLSU <compare characters less update>

p F8 See CEQD

CNED <compare characters not equal delete>

p F5 See CEQD

CNEU <compare characters not equal update>

p FD See CEQD

CPV <copy vector>

```
v   95E0

also represents CPVA, CPVN

See aAPIX:        current vector_A by stride_A
                  current vector_B by stride_B
See aFOP:         fetch via current vector_B
See aSOP:         store via current vector_A
See aVOAV:        validate vector_A:IWDD, vector_B:{IWDD,opnd}
                           length, stride_A, stride_B

Inv Arg/Par:      vector_A is sp and vector_B is dp or vice versa
```

CPVA <copy vector absolute>

```
v   95E2                                                    See CPV
```

CPVD <copy vector double>

```
v   95E4

See aAPIX:        current vector_A by stride_A
                  current vector_B by stride_B
See aFOP:         fetch via current vector_B
See aSOP:         store via current vector_A
See aVOAV:        validate vector_A:I2DD, vector_B:IWDD,
                           length, stride_A, stride_B
```

CPVN <copy vector negated>

```
v   95E1                                                    See CPV
```

CPVS <copy vector single>

```
v   95E3

See aAPIX:        current vector_A by stride_A
                  current vector_B by stride_B
See aFOP:         fetch via current vector_B
See aSOP:         fetch via current vector_A
See aVOAV:        validate vector_A:I1DD, vector_B:IWDD,
                           length, stride_A, stride_B

Vect Arith Exc:   RS(ND(B(i))) = TooBig              [Exponent Overflow]
                  RS(ND(B(i))) = TooSmall           [Exponent Underflow]
                  NS(RS(ND(B(i)))) = TooSmall       [Exponent Underflow]
```

CSCP <communicate with System Control Processor> (restricted)

```
v   959F
Unimplemented Op:  (unconditional)
```

DBCD <dynamic binary convert to decimal>

```
v   957F

Int Overflow:      RId(B) = TooBig
Int Arg/Par:       N (TOS) not operand or RaI(N) not {0 to 24}
                   B (TOS2) not operand
Uninit Arg:        TOS or TOS2 tag in {6, 4}
```

DBFL <dynamic branch false>

```
p   A8                                                    See DBUN
```

DBRS <dynamic bit reset>

```
p   9F
also represents DBST

Inv Arg/Par:       Db (TOS) not operand or RaI(Db) not {0 to 47}
Uninit Arg:        Db tag in {6, 4}
```

DBST <dynamic bit set>

```
p   97                                                    See DBRS
```

DBTR <dynamic branch true>

```
p   A9                                                    See DBUN
```

DBUN <dynamic branch unconditional>

```
p   AA
also represents DBFL & DBTR

See aEIRW:         evaluate SIRW
See aPRCW:         distribute branch-dest PCW code-stream pointer

Bounds Error:      (branch-dest opnd.dyn_pwi)mod 2**14 >= ActSegLen(PSN)
Inv Arg/Par:       branch-dest (TOS) not {operand, PCW, SIRW}
                   RaI (branch-dest operand) not {-2**39+1 to +2**39-1}
Inv Target:        SIRW ⟶▸ not PCW
Uninit Arg:        branch-dest (TOS) tag in {6, 4}
```

DEXI <disable external interrupts>

```
v   9547                                                  No interrupts
```

DFTR <dynamic field transfer>

```
p   99

Inv Arg/Par:      Len (TOS) not operand or RaI(Len) not {0 to 48}
                  Sb  (TOS2) not operand or RaI(Sb) not {0 to 47}
                  Db  (TOS3) not operand or RaI(Db) not {0 to 47}
Uninit Arg:       Len or Sb or Db tag in {6, 4}
```

DINS <dynamic field insert>

```
p   9D

Inv Arg/Par:      Len (TOS2) not operand or RaI(Len) not {0 to 48}
                  Db  (TOS3) not operand or RaI(Db) not {0 to 47}
Uninit Arg:       Len or Db tag in {6, 4}
```

DISO <dynamic field isolate>

```
p   9B

Inv Arg/Par:      Len (TOS) not operand or RaI(Len) not {0 to 48}
                  Sb  (TOS2) not operand or RaI(Sb) not {0 to 47}
Uninit Arg:       Len or Sb tag in {6, 4}
```

DIVD <divide>

```
p   83

Arith Overflow:   R(x/y) = TooBig                    [Exponent Overflow]
                  divisor (TOS) is 0                    [Divide by Zero]
Arith Underflow:  R(x/y) = TooSmall                 [Exponent Underflow]
                  R(x/y) < > R*(x/y)                    [Precision Loss]
                  N(R(x/y)) = TooSmall but R = R* < > 0      [spurious]
Inv Arg/Par:      x (TOS2) not operand or y (TOS) not operand
Uninit Arg:       TOS or TOS2 tag in {6, 4}
```

DLET <delete top of stack>

```
p   B5                                                  No interrupts
```

DOT <dot (inner) product - single>

```
v   95C4

See aAPIX:        current vector_A by stride_A
                  current vector_B by stride_B
See aFOP:         fetch via current vector_A or vector_B
See aVOAV:        validate vector_A:I1DD, vector_B:I1DD, length,
                           stride_A, stride_B, sum:sp operand

Vect Arith Exc:   R(A(i)*B(i)) = TooBig              [Exponent Overflow]
                  R(A(i)*B(i)) = TooSmall           [Exponent Underflow]
                  R(A(i)*B(i)) < > R*(A(i)*B(i))       [Precision Loss]
                  N(R(A(i)*B(i))) = TooSmall
                    but R = R* < > 0                          [spurious]
                  R(sum+...) = TooBig               [Exponent Overflow]
                  N(R(sum+...)) = TooSmall                    [spurious]
```

DOTX <dot (inner) product - extended>

```
v   95C5

See aAPIX:        current vector_A by stride_A
                  current vector_B by stride_B
See aFOP:         fetch via current vector_A or vector_B
See aVOAV:        validate vector_A:IWDD, vector_B:IWDD, length,
                           stride_A, stride_B, sum:dp operand

Vect Arith Exc:   R(A(i)*B(i)) = TooBig             [Exponent Overflow]
                  R(A(i)*B(i)) = TooSmall           [Exponent Underflow]
                  R(A(i)*B(i)) < > R*(A(i)*B(i))       [Precision Loss]
                  N(R(A(i)*B(i))) = TooSmall
                      but R = R* < > 0                        [spurious]
                  R(sum+...) = TooBig               [Exponent Overflow]
                  N(R(sum+...)) = TooSmall                    [spurious]
```

DRNT <dynamic range test>

```
v   9583

Inv Arg/Par:      X (TOS3) or L (TOS2) or H (TOS) not operand
Uninit Arg:       X or L or H tag in {6, 4}
```

DSLF <dynamic scale left>

```
p   C1                                                    See DSRF
```

DSRF <dynamic scale right>

```
p   C7
also represents DSLF, DSRS

Int Overflow:     RId(argument to be scaled) = TooBig
Inv Arg/Par:      sf (TOS) not operand or RaI(sf) not {0 to 12}
                  argument to be scaled (TOS2) not operand
Uninit Arg:       TOS or TOS2 tag in {6, 4}
```

DSRR <dynamic scale right rounded>

```
p   C9

Int Overflow:     RId(argument to be scaled) = TooBig
                     (when scale factor = 0)
                  TId(argument to be scaled) = TooBig
                     (when scale factor > 0)
Inv Arg/Par:      sf (TOS) not operand or RaI(sf) not {0 to 12}
                  argument to be scaled (TOS2) not operand
Uninit Arg:       TOS or TOS2 tag in {6, 4}
```

DSRS <dynamic scale right save>

```
p   C5                                                    See DSRF
```

DSRT <dynamic scale right truncate>

```
p   C3

Int Overflow:     TId(argument to be scaled) = TooBig
Inv Arg/Par:      sf (TOS) not operand or RaI(sf) not {0 to 12}
                  argument to be scaled (TOS2) not operand
Uninit Arg:       TOS or TOS2 tag in {6, 4}
```

DUPL <duplicate top of stack>

```
p   B7                                                 No interrupts
```

EEXI <enable external interrupts>

```
v   9546                                               No interrupts
```

ENDE <end edit>

```
te  DE                                                 No interrupts
```

ENDF <end float>

```
te  D5 & MinusChar:8 & PlusChar:8                         See INSU
```

ENTR <enter>

```
p   AB

See aEIRW:          evaluate SIRW at F+1 or successor SIRW
See aEVCH:          display update
See aPRCW:          distribute entry PCW code-stream pointer

Missing PCW:        Stk[F+1] not {SIRW, PCW}
                    SIRW ⟶▸ not {SIRW, PCW}
Structure Error:    Stk[F] not inactive MSCW
                    Stk[F].history_link_I >= 2**13 (frame overflow)
```

EQUL <equal to>

```
p   8C
also represents GREQ, GRTR, LESS, LSEQ, NEQL

Inv Arg/Par:        TOS not operand or TOS2 not operand
Uninit Arg:         TOS or TOS2 tag in {6, 4}
```

EVAL <evaluate>

```
p   AC

See aACCE:          SIRW ⟶▸ PCW
See aEIRW:          evaluate SIRW

Inv Arg/Par:        reference (TOS) not {SIRW, IWDD}
Uninit Arg:         TOS tag in {6, 4}
```

EXCH <exchange top of stack>

```
p   B6                                                  No interrupts
```

EXIT <exit>

```
p   A3
also represents RETN & REXI, RTN2
also partly represents aMVST

See aEVCH:          display update
See aPRCW:          distribute RCW code-stream pointer

Block Exit:         MSCW.block_exit=1
Structure Error:    Stk[D[LL]+1] not RCW
                    Stk[D[LL]] not entered MSCW
                    with i initially = D[LL],
                      while (i = D[LL]) or (Stk[i] not entered MSCW):
                         Stk[i] not MSCW (inactive or entered) or
                         MSCW.history_link_I >= 2**13 (frame o' overflow)
                         i:=i-(history link)
```

EXPU <execute single edit operator, single pointer update>

```
p   DD

See aPOAV:        validate dest:pointer, length:MaxInx

Undefined Op:     next operator is ENDE, ENDF, or INSC
                  next operator is MCHR, MVNU, MINS, MFLT,
                    SFSC, or SRSC (which requires source)
```

EXSD <execute single edit operator delete>

```
p   D2
also represents EXSU

See aPOAV:        validate dest:pointer, source:{pointer,opnd},
                          length:MaxInx

Inv Arg/Par:      source is EBCDIC and dest is hex or vice versa
Undefined Op:     next operator is ENDE, ENDF, or INSC
```

EXSU <execute single edit operator update>

```
p   DA                                                    See EXSD
```

FLTR <field transfer>

```
p   98 & Db:8 & Sb:8 & Len:8                          No interrupts
```

FMN <find minimum>

```
v   95CA
also represents FMX, FMXA

See aAPIX:        current vector_A by stride_A
See aFOP:         fetch via current vector_A
See aVOAV:        validate vector_A:IWDD, length, stride_A,
                          X:sp operand, V:operand

Inv Arg/Par:      vector_A is sp and V is dp or vice versa
```

FMX <find maximum>

```
v   95C9                                                  See FMN
```

FMXA <find maximum absolute>

```
v   95CB                                                  See FMN
```

GATH <gather>

v 95C1

See aAPIX:	current vector_A by stride_A vector_B argument by referent(vector_I) current vector_I by +1
See aFOP:	fetch via current vector_B or vector_I
See aSOP:	store via current vector_A
See aVOAV:	validate vector_A:IWDD, vector_B:IWDD, length, stride_A, vector_I:I1DD
Inv Arg/Par:	vector_A is sp and vector_B is dp or vice versa
Inv Target:	current vector_I ⟶▸ not single_integer |vector_I single_integer| > MaxInx

GESZ <get element_size>

v 95CF

Inv Arg/Par:	TOS not touched DD
Uninit Arg:	TOS tag in {6,4}

GINX <get index>

v 95CE

Inv Arg/Par:	TOS not pointer
Structure Error:	ASD3 for paged indexed DD is not unpaged indexed DD (not checked if page index known)
Uninit Arg:	TOS tag in {6, 4}

GLEN <get length>

v 95CD

Inv Arg/Par:	TOS not {WordDD, CharDD}
Structure Error:	unpaged or unindexed DD ⟶ not ASD2 ASD3 for paged indexed DD is not unpaged indexed DD (not checked if page index known) paged indexed DD PageTable.ASD_ref ⟶ not ASD2
Uninit Arg:	TOS tag in {6, 4}

GREQ <greater than or equal to>

p 89 See EQUL

GRTR <greater than>

p 8A See EQUL

HALT <unconditional processor halt> (restricted)

```
v   95DF                                                    No interrupts
```

ICLD <input convert left-signed delete>

```
v   9575                                                    See ICUD
```

ICRD <input convert right-signed delete>

```
v   9576                                                    See ICUD
```

ICUD <input convert unsigned delete>

```
p   A4
also represents ICLD, ICRD, ICVD, ICVU

See aCHAR:          access word via source ICDD
See aPOAV:          validate source:{pointer,opnd}, length:23

Pack Unindexed DD: source is unpaged unindexed copy DD
```

ICVD <input convert delete>

```
p   CA                                                      See ICUD
```

ICVU <input convert update>

```
p   CB                                                      See ICUD
```

IDIV <integer divide>

```
p   84
also partly represents RDIV

Arith Overflow:   divisor (TOS) is 0                  [Divide by Zero]
Int Overflow:     x and y are sp and |x DIV y| >= 2**39
                  x or  y is  dp and |x DIV y| >= 2**78
Inv Arg/Par:      x (TOS2) not operand or y (TOS) not operand
Uninit Arg:       TOS or TOS2 tag in {6, 4}
```

IMKS <insert mark stack>

```
p   CF                                                      No interrupts
```

INDX <index>

p A6

The representability interrupts associated with applying index to a DD are shown under aAPIX.

See aAPIX:	DD argument by RaI(index argument)
See aEIRW:	evaluate SIRW
Bounds Error:	|RaI(index argument)| > MaxInx new word index > ActSegLen new word index = ActSegLen and new char index > 0
Inv Arg/Par:	not (TOS in {WordDD, CharDD, SIRW} and TOS2 is operand or TOS is operand and TOS2 in {WordDD, CharDD, SIRW}) TOS or TOS2 is reindex-disabled IWDD or ICDD EBCDIC ICDD.char_index > 5 hex ICDD.char_index > 11
Inv Target:	SIRW ⟶⟶ not {SIRW, WordDD, CharDD, untouched DD} SIRW ⟶⟶ reindex-disabled IWDD or ICDD
Structure Error:	DD ⟶ not ASD1/ASD2
Touch Reference:	SIRW ⟶⟶ untouched DD
Uninit Arg:	TOS or TOS2 tag in {6, 4}

INOP <insert overpunch>

e D8 — See INSG

INSC <insert conditional>

te DD & Length:8 & ZeroChar:8 & NonZeroChar:8 — See INSU

INSG <insert display sign>

e D9 & MinusChar:8 & PlusChar:8
also represents INOP

Interrupts same as INSU, plus:

Inv Arg/Par: dest is hex

INSR <field insert>

p 9C & Db:8 & Len:8 — No interrupts

INSU <insert unconditional>

```
se  DC & Char:8
te  DC & Length:8 & Char:8
also represents ENDF, INSC
also partly represents INSG & INOP

See aCHAR:          access word via dest ICDD

Dest Boundary:      dest segment exhausted
Inv Access:         read_only destination pointer
```

ISOL <field isolate>

```
p   9A & Sb:8 & Len:8                                   No interrupts
```

JOIN <join two operands into double>

```
v   9542

Inv Arg/Par:        TOS not operand or TOS2 not operand
Uninit Arg:         TOS or TOS2 tag in {6, 4}
```

LAND <logical and>

```
p   90                                                  No interrupts
```

LEQV <logical equivalence>

```
p   93                                                  No interrupts
```

LESS <less than>

```
p   88                                                      See EQUL
```

LNMC <long name call>

```
v   958C & lambda:4 & delta:12                              See NAMC
```

LNOT <logical not>

```
p   92                                                  No interrupts
```

LOAD <load>

```
p   BD

See aFOP:          fetch via SIRW or IWDD

Inv Arg/Par:       reference (TOS) not {SIRW, IWDD}
Inv Target:        SIRW or I1DD ⟶▸ not {SIRW, DD, Data Word}
Touch Load:        SIRW or I1DD ⟶▸ untouched DD
Uninit Arg:        TOS tag in {6, 4}
```

LODT <load transparent>

```
p   BC
v   95BC

See aEIRW:         evaluate SIRW
See aEIXD:         evaluate IWDD

Inv Arg/Par:       ref (TOS) not {SIRW, IWDD, ASRW,
                                  operand in {+0 to 2**29-1}}
Uninit Arg:        TOS in tag {6, 4}
```

LOG2 <leading one test>

```
v   958B                                                  No interrupts
```

LOK <lock interlock>

```
v   95B0

See aILRE:         interlock reference evaluations

Locking:           interlock status not Free
```

LOR <logical or>

```
p   91                                                    No interrupts
```

LPOW <load POW> (restricted)

```
v   954D

See aEIRW:         evaluate SIRW
See aEIXD:         evaluate IWDD

Activate Object:   reference ⟶▸ operand
Inv Arg/Par:       reference (TOS) not {SIRW, IWDD}
Inv Target:        SIRW ⟶▸ not {SIRW, IWDD, PCW, operand, POW}
                   IWDD ⟶▸ not {operand, POW}
Uninit Arg:        TOS tag in {6, 4}
```

LSEQ <less than or equal to>

```
p   8B                                                    See EQUL
```

LT8 <insert 8-bit literal>

```
p   B2 & constant:8                                       No interrupts
```

LT16 <insert 16-bit literal>

```
p   B3 & constant:16                                      No interrupts
```

LT48 <insert 48-bit literal>

```
p   BE & word-aligned constant:48                         No interrupts
```

LVLC <long value name call>

```
v   958D & lambda:4 & delta:12                            See VALC
```

MCHR <move characters>

```
se  D7
te  D7 & Length:8
also represents MFLT, MINS, MVNU
```

See aCHAR: access word via source or dest ICDD

```
Dest Boundary:    dest segment exhausted
Inv Access:       read_only destination pointer
Source Boundary:  source segment exhausted
```

MFLT <move with float>

```
se  D1 & ZeroChar:8 & MinusChar:8 & PlusChar:8
te  D1 & Length:8 & ZeroChar:8 & MinusChar:8 & PlusChar:8    See MCHR
```

MINS <move with insert>

```
se  D0 & ZeroChar:8
te  D0 & Length:8 & ZeroChar:8                            See MCHR
```

MKSN <mark-stack bound to name-call>

```
p   DF                                                    No interrupts
```

MKST <mark stack>

```
p   AE                                                    No interrupts
```

MPCW <make PCW>

```
p   BF & word-aligned SkeletonPCW:48

See aEAC:          evaluate (sdll,sdi) from parameter

Inv Target:        (sdll,sdi) ⟶▸ not {CSD, untouched CSD}
Touch CSD:         (sdll,sdi) ⟶▸ untouched CSD
```

MULT <multiply>

```
p   82
also represents MULX

Arith Overflow:    R(x*y) = TooBig                    [Exponent Overflow]
Arith Underflow:   R(x*y) = TooSmall                 [Exponent Underflow]
                   R(x*y) < > R*(x*y)                   [Precision Loss]
                   N(R(x*y)) = TooSmall but R = R* < > 0      [spurious]
Inv Arg/Par:       x (TOS2) not operand or y (TOS) not operand
Uninit Arg:        TOS or TOS2 tag in {6, 4}
```

MULX <extended multiply>

```
p   8F                                                    See MULT
```

MVNU <move numeric unconditional>

```
se  D6
te  D6 & Length:8                                         See MCHR
```

NAMC <name call>

```
p   01:2 & variable-fence address couple:14 (two syllables total)
also represents LNMC

See aEAC:          evaluate parameter
```

NEQL <not equal to>

```
p   8D                                                    See EQUL
```

NOOP <no operation>

```
p   FE
v   95FE                                              No interrupts
```

NORM <normalize>

v 958E

```
Arith Overflow:   N(x) = TooSmall                    [Exponent Underflow]
Inv Arg/Par:      x (TOS) not operand
Uninit Arg:       TOS tag in {6, 4}
```

NORX <disable reindex of reference>

v 957C

```
Inv Arg/Par:      TOS not indexed DD
Uninit Arg:       TOS tag in {6, 4}
```

NOWR <disable write via reference>

v 957D

```
Inv Arg/Par:      TOS not touched DD
Uninit Arg:       TOS tag in {6, 4}
```

NTGD <integerize double precision rounded>

v 9587

```
Int Overflow:     RId(x) = TooBig
Inv Arg/Par:      x (TOS) not operand
Uninit Arg:       TOS tag in {6, 4}
```

NTGR <integerize rounded>

p 87

```
Int Overflow:     RaI(x) = TooBig
Inv Arg/Par:      x (TOS) not operand
Uninit Arg:       TOS tag in {6, 4}
```

NTIA <integerize truncated>

```
p   86

Int Overflow:     TI(x) = TooBig
Inv Arg/Par:      x (TOS) not operand
Uninit Arg:       TOS tag in {6, 4}
```

NTTD <integerize double precision truncated>

```
v   9586

Int Overflow:     TId(x) = TooBig
Inv Arg/Par:      x (TOS) not operand
Uninit Arg:       TOS tag in {6, 4}
```

NVLD <invalid operator>

```
p   FF
v   95FF

Invalid Operator: (unconditional)
```

NXLN <index and load name>

```
p   A5

See aAPIX:        SingleDD argument by RaI(index argument)
See aEIRW:        evaluate SIRW
See aEIXD:        evaluate final indexed SingleDD

Bounds Error:     |RaI(index argument)| > MaxInx
Inv Arg/Par:      not (    TOS in {SingleDD, SIRW} and
                           TOS2 is operand
                       or TOS is operand and
                           TOS2 in {SingleDD, SIRW}    )
                  TOS or TOS2 is reindex-disabled I1DD
Inv Target:       SIRW ⟶▸ not {SIRW, SingleDD, untouched DD}
                  SIRW ⟶▸ reindex-disabled I1DD
                  SingleDD[index argument] ⟶▸ not {untouched DD,
                                                   unindexed DD}

Touch Load:       SingleDD[index argument] ⟶▸ untouched DD
Touch Reference:  SIRW ⟶▸ untouched DD
Uninit Arg:       TOS or TOS2 tag in {6, 4}
```

NXLV <index and load value>

```
p   AD

See aAPIX:        WordDD argument by RaI(index argument)
See aEIRW:        evaluate SIRW
See aFOP:         fetch via final indexed WordDD

Bounds Error:     |RaI(index argument)| > MaxInx
Inv Arg/Par:      not (    TOS in {WordDD, SIRW} and
                           TOS2 is operand
                       or TOS is operand and
                           TOS2 in {WordDD, SIRW}    )
                  TOS or TOS2 is reindex-disabled IWDD
Inv Target:       SIRW ⟶▸ not {SIRW, WordDD, untouched DD}
                  SIRW ⟶▸ reindex-disabled IWDD
Touch Reference:  SIRW ⟶▸ untouched DD
Uninit Arg:       TOS or TOS2 tag in {6, 4}
```

OCRX <occurs index>

```
v   9585

Bounds Error:     RaI(index argument) not {1 to ICW.ICW_limit}
Inv Arg/Par:      ICW (TOS) not sp opnd or index (TOS2) not opnd
Uninit Arg:       TOS or TOS2 tag in {6, 4}
```

ONE <insert literal one>

```
p   B1                                                  No interrupts
```

OVRD <overwrite delete>

```
p   BA
also represents OVRN and RDLK

See aEIRW:        evaluate SIRW
See aEIXD:        evaluate I1DD

Inv Access:       read_only I1DD
Inv Arg/Par:      reference (TOS) not {SIRW, I1DD, ASRW}
Uninit Arg:       reference tag in {6, 4}
```

OVRN <overwrite nondelete>

```
p   BB                                                  See OVRD
```

PACD <pack delete>

```
p   D1                                                  See PKUD
```

PACU <pack update>

```
p   D9                                                    See PKUD
```

PKLD <pack left-signed delete>

```
v   9573                                                  See PKUD
```

PKRD <pack right-signed delete>

```
v   9574                                                  See PKUD
```

PKUD <pack unsigned delete>

```
v   9572
also represents PACD, PACU, PKLD, PKRD

See aCHAR:          access word via source ICDD
See aPOAV:          validate source:{pointer,opnd}, length:24

Pack Unindexed DD: source is unpaged unindexed copy DD
```

POLY <polynomial recurrence>

```
v   95C7

See aAPIX:        current vector_A by stride_A
                  current vector_B by stride_B
See aFOP:         fetch via current vector_A or vector_B
See aVOAV:        validate vector_A:IWDD, vector_B:{IWDD,opnd},
                           length, stride_A, stride_B, Z:operand

Inv Arg/Par:      vector_A or vector_B is dp and Z is sp
Vect Arith Exc:   R(B(i)*Z) = TooBig                 [Exponent Overflow]
                  R(B(i)*Z) = TooSmall              [Exponent Underflow]
                  R(B(i)*Z) < > R*(B(i)*Z)             [Precision Loss]
                  N(R(B(i)*Z)) = TooSmall
                       but R = R* < > 0                        [spurious]
                  R(A(i)+...) = TooBig              [Exponent Overflow]
                  N(R(A(i)+...)) = TooSmall                  [spurious]
```

PUSH <push working stack onto activation record>

```
p   B4                                                  No interrupts
```

RASD <read ASD-table word> (restricted)

```
v   95B1
also represents RAST, WASD, WAST
Inv Arg/Par:      specifier (TOS) not operand or |RaI(TOS)| > MaxInx
                  ASD_num (TOS2) not operand or |RaI(TOS2)| > MaxInx
Uninit Arg:       TOS or TOS2 tag in {6, 4}
```

RAST <RASD trapping synonym> (restricted)

```
v   95D6                                                    See RASD
```

RDIV <remainder divide>

```
p   85
Interrupts same as IDIV, plus:
Arith Underflow:  N(x MOD y) = TooSmall                     [spurious]
```

RDLK <read and lock>

```
v   95BA                                                    See OVRD
```

RETN <return>

```
p   A7                                                      See EXIT
```

REXI <return/exit from interrupt> (restricted)

```
v   95F7                                                    See EXIT
```

RNGT <range test>

```
v   9582 & l:8 & h:8
Inv Arg/Par:      X (TOS) not operand
Uninit Arg:       TOS tag in {6, 4}
```

ROFF <read and reset overflow flip-flop>

```
p   D7                                                  No interrupts
```

RPRR <read processor register>

v 95B8

Inv Arg/Par: reg-id (TOS) not operand or RAI(reg-id) not in {0-15, 37-38, 52-53}

Uninit Arg: TOS tag in {6, 4}

RSDN <rotate stack down>

v 95B7 No interrupts

RSNR <read SNR>

v 9581 No interrupts

RSQR <real square root>

v 9550

Inv Arg/Par: x (TOS) not operand or x < 0

Uninit Arg: TOS tag in {6, 4}

RSTF <reset float flip-flop>

e D4 No interrupts

RSUP <rotate stack up>

v 95B6 No interrupts

RTAG <read tag>

v 95B5 No interrupts

RTFF <read true-false flip-flop>

p DE No interrupts

RTN2 <return two items>

v 95AE See EXIT

RTOD <read time of day clock>

v 95A7 No interrupts

RUNI <indicate running> (restricted)

```
v   9541                                                    No interrupts
```

SAME <logical equality>

```
p   94                                                      No interrupts
```

SCAT <scatter>

```
v   95C0

See aAPIX:      vector_A argument by referent(vector_I)
                current vector_B by stride_B
                current vector_I by +1
See aFOP:       fetch via current vector_B or vector_I
See aSOP:       store via current vector_A
See aVOAV:      validate vector_A:IWDD, vector_B:{IWDD,opnd},
                         length, vector_I:I1DD, stride_B

Inv Arg/Par:    vector_A is sp and vector_B is dp or vice versa
Inv Target:     current vector_I ⟶▸ not single_integer
                |vector_I single_integer| > MaxInx
```

SCLF <scale left>

```
p   C0 & ScaleFactor:8                                      See SCRF
```

SCRF <scale right final>

```
p   C6 & ScaleFactor:8
also represents SCLF, SCRS

Int Overflow:   RId(argument to be scaled) = TooBig
Int Arg/Par:    argument to be scaled (TOS) not operand
                ScaleFactor > 12
Uninit Arg:     TOS tag in {6, 4}
```

SCRR <scale right rounded>

```
p   C8 & ScaleFactor:8

Int Overflow:   RId(argument to be scaled) = TooBig
                   (when scale factor = 0)
                TId(argument to be scaled) = TooBig
                   (when scale factor > 0)
Inv Arg/Par:    argument to be scaled (TOS) not operand
                ScaleFactor > 12
Uninit Arg:     TOS tag in {6, 4}
```

SCRS <scale right save>

```
p   C4 & ScaleFactor:8                                          See SCRF
```

SCRT <scale right truncate>

```
p   C2 & ScaleFactor:8

Int Overflow:     TId(argument to be scaled) = TooBig
Int Arg/Par:      argument to be scaled (TOS) not operand
                  ScaleFactor > 12
Uninit Arg:       TOS tag in {6, 4}
```

SEDW <set element_size to double words>

```
v   95A3                                                        See SESW
```

SEQ <sequential recurrence>

```
v   95C6

See aAPIX:        current vector_A by stride_A
                  current vector_B by stride_B
See aFOP:         fetch via current vector_B
See aSOP:         store via current vector_A
See aVOAV:        validate vector_A:IWDD, vector_B:{IWDD,opnd},
                           length, stride_A, stride_B, Z:operand

Inv Arg/Par:      vector_A is sp and vector_B is dp
                  vector_A is sp and Z is dp or vice versa
Vect Arith Exc:   R(Z+B(i)) = TooBig                [Exponent Overflow]
                  N(R(Z+B(i))) = TooSmall                       [spurious]
```

SEQD <scan while equal delete>

```
v   95F4
also represents SEQU, SGED, SGEU, SGTD, SGTU, SLED,
                SLEU, SLSD, SLSU, SNED, SNEU

See aCHAR:        access word via source ICDD
See aPOAV:        validate source:{pointer,opnd},
                           length:MaxInx, delim:sp operand
```

SEQU <scan while equal update>

```
v   95FC                                                        See SEQD
```

SESW <set element_size to single words>

```
v   95A2
also represents SEDW

Inv Arg/Par:       TOS not {WordDD, CharDD}
                   ICDD.char_index < > 0
                   pointer reindex disabled
Structure Error:   ASD3 for paged indexed DD is not unpaged indexed DD
                   (not checked if page index known)
Uninit Arg:        TOS in tag {6, 4}
```

SE4C <set element_size to hex characters>

```
v   95A0                                                    See SE8C
```

SE8C <set element_size to EBCDIC characters>

```
v   95A1
also represents SE4C

See aAPIX:         argument pointer by 0  (after element_size change)

Bounds Error:      IWDD.index >= 2**16 (implied by aAPIX)
                   unpaged pointer word index > ActSegLen
Inv Arg/Par:       TOS not {WordDD, CharDD}
                   ICDD.char_index < > 0
                   pointer reindex disabled
Structure Error:   unpaged pointer ⟶ not ASD1/ASD2
Uninit Arg:        TOS tag in {6, 4}
```

SFDC <skip forward destination characters>

```
se  DA
te  DA & Length:8

See aAPIX:         current dest ICDD by +max(length,0)

Bounds Error:      new word index > ActSegLen
                   new word index = ActSegLen and
                       new char index > 0
Structure Error:   ICDD ⟶ not ASD1/ASD2

Bounds and Structure errors not reported by EXSD SFDC.
```

SFSC <skip forward source characters>

```
se  D2
te  D2 & Length:8

See aAPIX:        current source ICDD by +max(length,0)

Bounds Error:     new word index > ActSegLen
                  new word index = ActSegLen and
                      new char index > 0
Structure Error:  ICDD ⟶ not ASD1/ASD2

Bounds and Structure errors not reported by EXSD SFSC.
```

SGED <scan while greater or equal delete>

```
v   95F1                                                    See SEQD
```

SGEU <scan while greater or equal update>

```
v   95F9                                                    See SEQD
```

SGTD <scan while greater delete>

```
v   95F2                                                    See SEQD
```

SGTU <scan while greater update>

```
v   95FA                                                    See SEQD
```

SHOW <primitive display> (restricted)

```
v   95DE
Unimplemented Op:  (unconditional)
```

SISO <string isolate>

```
p   D5

See aCHAR:        access word via source ICDD
See aPOAV:        validate source:{pointer,opnd},
                           length:(if source is hex then 24 else 12)
```

SLED <scan while less or equal delete>

```
v   95F3                                                    See SEQD
```

SLEU <scan while less or equal update>

```
v   95FB                                                    See SEQD
```

SLSD <scan while less delete>

v 95F0 See SEQD

SLSU <scan while less update>

v 95F8 See SEQD

SNED <scan while not equal delete>

v 95F5 See SEQD

SNEU <scan while equal update>

v 95FD See SEQD

SNGL <set to single precision rounded>

p CD

Arith Overflow:	RS(ND(x)) = TooBig	[Exponent Overflow]
Arith Underflow:	RS(ND(x)) = TooSmall	[Exponent Underflow]
	NS(RS(ND(x))) = TooSmall	[Exponent Underflow]
Inv Arg/Par:	x (TOS) not operand	
Uninit Arg:	TOS in tag {6, 4}	

SNGT <set to single precision truncated>

p CC

Arith Overflow:	TS(ND(x)) = TooBig	[Exponent Overflow]
Arith Underflow:	TS(ND(x)) = TooSmall	[Exponent Underflow]
	NS(TS(ND(x))) = TooSmall	[Exponent Underflow]
Inv Arg/Par:	x (TOS) not {operand, WordDD}	
	IWDD reindex disabled	
Uninit Arg:	TOS in tag {6, 4}	

SPLT <split operand into two singles>

v 9543

Inv Arg/Par:	TOS not operand
Uninit Arg:	TOS in tag {6, 4}

SPRR <set processor register> (restricted)

v 95B9

Inv Arg/Par: reg-val (TOS) not operand
reg-id (TOS2) not operand or RaI(reg-id) in {0-15, 37, 53}
Uninit Arg: TOS or TOS2 tag in {6, 4}

SRCH <masked search for equal>

v 95BE

It is convenient to define SRCH as a variant of BMS in which the length argument is implicitly one more than the initial virtual index of the domain argument. The domain is exhausted when the length is counted to zero.

See aAPIX: current domain by -1
See aEIXD: evaluate current domain

Inv Arg/Par: domain (TOS) not I1DD
domain reindex disabled
Uninit Arg: domain tag in {6, 4}

SRDC <skip reverse destination characters>

se DB
te DB & Length:8

See aAPIX: current dest ICDD by -max(length,0)

Structure Error: ICDD ⟶ not ASD1/ASD2

SRSC <skip reverse source characters>

se D3
te D3 & Length:8

See aAPIX: current source ICDD by -max(length,0)

Structure Error: ICDD ⟶ not ASD1/ASD2

STAG <set tag>

v 95B4

Inv Arg/Par: tag value (TOS) not operand
Uninit Arg: TOS in tag {6, 4}

STFF <stuff>

```
p   AF                                                    No interrupts
```

STOD <store delete>

```
p   B8
also represents STON

See aACCE:        SIRW ⟶▸ PCW
See aEIRW:        evaluate SIRW
See aSOP:         store via SIRW or IWDD

Inv Arg/Par:      not (    TOS in {SIRW, IWDD} and TOS2 is operand
                       or TOS is operand and TOS2 in {SIRW, IWDD} )
Inv Target:       SIRW ⟶▸ not {SIRW, IWDD, PCW, untouched DD,
                              Data Word}
Touch Reference:  SIRW ⟶▸ untouched DD
Uninit Arg:       TOS or TOS2 tag in {6, 4}
```

STON <store nondelete>

```
p   B9                                                          See STOD
```

STOP <unconditional processor halt> (restricted)

```
v   95BF                                                  No interrupts
```

SUBT <subtract>

```
p   81

Arith Overflow:   R(x-y) = TooBig                  [Exponent Overflow]
Arith Underflow:  N(R(x-y)) = TooSmall                        [spurious]
Inv Arg/Par:      x (TOS2) not operand or y (TOS) not operand
Uninit Arg:       TOS or TOS2 tag in {6, 4}
```

SUM <summation over vector>

```
v   95C2
also represents SUMA

See aAPIX:        current vector_A by stride_A
See aFOP:         fetch via current vector_A
See aVOAV:        validate vector_A:IWDD, length, stride_A,
                           sum:operand

Inv Arg/Par:      current vector_A is dp and sum is sp
Vect Arith Exc:   R(sum+A(i)) = TooBig             [Exponent Overflow]
                  N(R(sum+(A(i))) = TooSmall                  [spurious]
```

SUMA <summation over vector, absolute>

v 95C3 See SUM

SWFD <scan while false delete>

v 95D4
also represents SWFU, SWTD, SWTU

See aAPIX: set by WordIndex(c)
See aCHAR: access word via source ICDD
See aEIXD: evaluate set[WordIndex(c)], if length > 0
See aPOAV: validate source:{pointer,opnd},
length:MaxInx, set:I1DD

Inv Target: set[WordIndex(c)] ⟶▸ not operand

SWFU <scan while false update>

v 95DC See SWFD

SWTD <scan while true delete>

v 95D5 See SWFD

SWTU <scan while true update>

v 95DD See SWFD

SXSN <set external sign flip-flop>

p D6

Inv Arg/Par: TOS not operand
Uninit Arg: TOS tag in {6, 4}

TEED <table enter edit delete>

```
p   DO
also represents TEEU

See aPOAV:        validate dest:pointer, source:{pointer,opnd},
                           edtab:indexed DD

Absent Code:      edtab
Bounds Error:     edtab.ewi >= ActSegLen(table)
                  edtab.esi > 5
Inv Arg/Par:      source is EBCDIC and dest is hex or vice versa
Structure Error:  edtab ⟶ not ASD1/ASD2
```

TEEU <table enter edit update>

```
p   D8                                                    See TEED
```

TEQD <transfer while equal delete>

```
p   E4
also represents TEQU, TGED, TGEU, TGTD, TGTU, TLED,
                TLEU, TLSD, TLSU, TNED, TNEU

See aCHAR:        access word via source or dest ICDD
See aPOAV:        validate dest:pointer, source:{pointer,opnd},
                           length:MaxInx, delim:sp operand

Dest Boundary:    dest segment exhausted
Inv Access:       read_only destination pointer
Inv Arg/Par:      source is EBCDIC and dest is hex or vice versa
Source Boundary:  source element exhausted
```

TEQU <transfer while equal update>

```
p   EC                                                    See TEQD
```

TGED <transfer while greater or equal delete>

```
p   E1                                                    See TEQD
```

TGEU <transfer while greater or equal update>

```
p   E9                                                    See TEQD
```

TGTD <transfer while greater delete>

```
p   E2                                                    See TEQD
```

TGTU <transfer while greater update>

p EA — See TEQD

TLED <transfer while less or equal delete>

p E3 — See TEQD

TLEU <transfer while less or equal update>

p EB — See TEQD

TLSD <transfer while less delete>

p E0 — See TEQD

TLSU <transfer while less update>

p E8 — See TEQD

TNED <transfer while not equal delete>

p E5 — See TEQD

TNEU <transfer while not equal update>

p ED — See TEQD

TRNS <translate>

v 95D7

See aAPIX:	trtab by WordIndex(c)
See aCHAR:	access word via source or dest ICDD
See aEIXD:	evaluate trtab[WordIndex(c)], if length > 0
See aPOAV:	validate dest:pointer, source:{pointer,opnd}, length:MaxInx, trtab:I1DD
Dest Boundary:	dest segment exhausted
Inv Access:	read_only destination pointer
Inv Target:	trtab[WordIndex(c)] ⟶▸ not operand
Source Boundary:	source segment exhausted

TUND <transfer characters unconditional delete>

```
p   E6
also represents TUNU

See aCHAR:        access word via source or dest ICDD
See aPOAV:        validate dest:pointer, source:{pointer,opnd},
                           length:MaxInx

Dest Boundary:    dest segment exhausted
Inv Access:       read_only destination pointer
Inv Arg/Par:      source is EBCDIC and dest is hex or vice versa
Source Boundary:  source segment exhausted
```

TUNU <transfer characters unconditional update>

```
p   EE                                                    See TUND
```

TWFD <transfer while false delete>

```
v   95D2
also represents TWFU, TWTD, TWTU

See aAPIX:        set by WordIndex(c)
See aCHAR:        access word via source or dest ICDD
See aEIXD:        evaluate set[WordIndex(c)], if length > 0
See aPOAV:        validate dest:pointer, source:{pointer,opnd},
                           length:MaxInx,  set:I1DD

Dest Boundary:    dest segment exhausted
Inv Access:       read_only destination pointer
Inv Arg/Par:      source is EBCDIC and dest is hex or vice versa
Inv Target:       set[WordIndex(c)] ⟶▸ not operand
Source Boundary:  source segment exhausted
```

TWFU <transfer while false update>

```
v   95DA                                                  See TWFD
```

TWOD <transfer words overwrite delete> (restricted)

```
p   D4

See aAPIX:        current source or dest pointer (as I1DD) by +1

See aEIXD:        evaluate current source or dest pointer, if length > 0
See aPOAV:        validate dest:{pointer,ASRW},
                           source:{pointer,ASRW,opnd},
                           length:MaxInx

Inv Access:       read_only destination pointer
```

TWSD <transfer words delete>

```
p   D3
also partly represents TWSU

See aAPIX:        current source or dest pointer (as I1DD) by +1
See aEIXD:        evaluate current source or dest pointer, if length > 0
See aPOAV:        validate dest:pointer, source:{pointer,opnd},
                           length:MaxInx

Dest Boundary:    dest segment exhausted
Inv Access:       read_only destination pointer
Inv Target:       dest pointer ⟶▸ not Data Word
                  source pointer ⟶▸ not operand
Source Boundary:  source segment exhausted
```

TWSU <transfer words update>

```
p   DB

Interrupts same as TWSD, plus:

Bounds Error:     updated source or destination ICDD has word_index
                       >= 2**16
```

TWTD <transfer while true delete>

```
v   95D3                                                    See TWFD
```

TWTU <transfer while true update>

```
v   95DB                                                    See TWFD
```

UNLK <unlock interlock>

```
v   95B2

See aILRE:        interlock reference evaluations

Unlocking:        interlock status not Lock_Uncontended
```

UPLD <unpack left-signed delete>

```
v   9570                                                    See UPUD
```

UPLU <unpack left-signed update>

```
v   9578                                                    See UPUD
```

UPRD <unpack right-signed delete>

v 9571 See UPUD

UPRU <unpack right-signed update>

v 9579 See UPUD

UPUD <unpack unsigned delete>

v 95D1
also represents UPLD, UPLU, UPRD, UPRU, UPUU, USND, USNU

See aCHAR: access word via dest ICDD
See aPOAV: validate dest:pointer, source:operand, length:24

Dest Boundary: dest segment exhausted
Inv Access: read_only destination pointer

UPUU <unpack unsigned update>

v 95D9 See UPUD

USND <unpack signed delete>

v 95D0 See UPUD

USNU <unpack signed update>

v 95D8 See UPUD

VALC <value call>

p 00:2 & variable-fence address couple:14 (two syllables total)
also represents LVLC

See aACCE: parameter or SIRW ⟶▸ PCW
See aFOP: fetch via parameter or subsequent reference

Inv Arg/Par: restart TOS not {SIRW, IWDD, operand}
Inv Target: parameter or SIRW ⟶▸
not {SIRW, IWDD, PCW, untouched DD, opnd}
Touch Reference: parameter or SIRW ⟶▸ untouched DD

VARI <introduce variant operator>

p 95 No interrupts

VMV <vector minus vector>

```
v   95E6

See aAPIX:        current vector_A by stride_A
                  current vector_B by stride_B
See aFOP:         fetch via current vector_A or vector_B
See aSOP:         store via current vector_A
See aVOAV:        validate vector_A:IWDD, vector_B:{IWDD,opnd},
                          length, stride_A, stride_B

Inv Arg/Par:      vector_A is sp and vector_B is dp
Vect Arith Exc:   R(A(i)-B(i)) = TooBig              [Exponent Overflow]
                  N(R(A(i)-B(i))) = TooSmall                  [spurious]
```

VMVN <vector minus vector negated>

```
v   95E7

See aAPIX:        current vector_A by stride_A
                  current vector_B by stride_B
See aFOP:         fetch via current vector_A or vector_B
See aSOP:         store via current vector_A
See aVOAV:        validate vector_A:IWDD, vector_B:{IWDD,opnd},
                          length, stride_A, stride_B

Inv Arg/Par:      vector_A is sp and vector_B is dp
Vect Arith Exc:   R(B(i)-A(i)) = TooBig              [Exponent Overflow]
                  N(R(B(i)-A(i))) = TooSmall                  [spurious]
```

VOV <vector over (divided by) vector>

```
v   95E9

See aAPIX:        current vector_A by stride_A
                  current vector_B by stride_B
See aFOP:         fetch via current vector_A or vector_B
See aSOP:         store via current vector_A
See aVOAV:        validate vector_A:IWDD, vector_B:{IWDD,opnd},
                          length, stride_A, stride_B

Inv Arg/Par:      vector_A is sp and vector_B is dp
Vect Arith Exc:   R(A(i)/B(i)) = TooBig              [Exponent Overflow]
                  R(A(i)/B(i)) = TooSmall            [Exponent Underflow]
                  R(A(i)/B(i)) < > R*(A(i)/B(i))        [Precision Loss]
                  N(R(A(i)/B(i))) = TooSmall
                      but R = R* < > 0                        [spurious]
                  B(i) is 0                             [Divide by Zero]
```

VPS <vector plus scalar>

```
v   95EB

See aAPIX:        current vector_A by stride_A
                  current vector_B by stride_B
See aFOP:         fetch via current vector_B
See aSOP:         store via current vector_A
See aVOAV:        validate vector_A:IWDD, vector_B:IWDD,
                           length, stride_A, stride_B, S:operand

Inv Arg/Par:      vector_A is sp and vector_B is dp
                  vector_A is sp and S is dp or vice versa
Vect Arith Exc:   R(B(i)+S) = TooBig                [Exponent Overflow]
                  N(R(B(i)+S)) = TooSmall                     [spurious]
```

VPV <vector plus vector>

```
v   95E5

See aAPIX:        current vector_A by stride_A
                  current vector_B by stride_B
See aFOP:         fetch via current vector_A or vector_B
See aSOP:         store via current vector_A
See aVOAV:        validate vector_A:IWDD, vector_B:{IWDD,opnd},
                           length, stride_A, stride_B

Inv Arg/Par:      vector_A is sp and vector_B is dp
Vect Arith Exc:   R(A(i)+B(i)) = TooBig             [Exponent Overflow]
                  N(R(A(i)+B(i))) = TooSmall                  [spurious]
```

VPVS <vector plus vector times scalar>

```
v 95ED

See aAPIX:        current vector_A by stride_A
                  current vector_B by stride_B
See aFOP:         fetch via current vector_A or vector_B
See aSOP:         store via current vector_A
See aVOAV:        validate vector_A:IWDD, vector_B:IWDD,
                           length, stride_A, stride_B, S:operand

Inv Arg/Par:      vector_A is sp and vector_B is dp
                  vector_A is sp and S is dp or vice versa
Vect Arith Exc:   R(B(i)*S) = TooBig                [Exponent Overflow]
                  R(B(i)*S) = TooSmall             [Exponent Underflow]
                  R(B(i)*S) < > R*(B(i)*S)             [Precision Loss]
                  N(R(B(i)*S)) = TooSmall
                     but R = R* < > 0                         [spurious]
                  R(A(i)+...) = TooBig              [Exponent Overflow]
                  N(R(A(i)+...)) = TooSmall                   [spurious]
```

VSPV <vector times scalar plus vector>

```
v   95EE

See aAPIX:        current vector_A by stride_A
                  current vector_B by stride_B
See aFOP:         fetch via current vector_A or vector_B
See aSOP:         store via current vector_A
See aVOAV:        validate vector_A:IWDD, vector_B:{IWDD,opnd}
                           length, stride_A, stride_B, S:operand

Inv Arg/Par:      vector_A is sp and vector_B is dp
                  vector_A is sp and S is dp or vice versa
Vect Arith Exc:   R(A(i)*S) = TooBig                [Exponent Overflow]
                  R(A(i)*S) = TooSmall              [Exponent Underflow]
                  R(A(i)*S) < > R*(A(i)*S)             [Precision Loss]
                  N(R(A(i)*S)) = TooSmall
                     but R = R* < > 0                          [spurious]
                  R(...+B(i)) = TooBig              [Exponent Overflow]
                  N(R(...+B(i))) = TooSmall                  [spurious]
```

VTS <vector times scalar>

```
v   95EC

See aAPIX:        current vector_A by stride_A
                  current vector_B by stride_B
See aFOP:         fetch via current vector_B
See aSOP:         store via current vector_A
See aVOAV:        validate vector_A:IWDD, vector_B:IWDD,
                           length, stride_A, stride_B, S:operand

Inv Arg/Par:      vector_A is sp and vector_B is dp
                  vector_A is sp and S is dp or vice versa
Vect Arith Exc:   R(B(i)*S) = TooBig                [Exponent Overflow]
                  R(B(i)*S) = TooSmall              [Exponent Underflow]
                  R(B(i)*S) < > R*(B(I)*S)             [Precision Loss]
                  N(R(B(i)*S)) = TooSmall
                     but R = R* < > 0                          [spurious]
```

VTV <vector times vector>

```
v   95E8

See aAPIX:        current vector_A by stride_A
                  current vector_B by stride_B
See aFOP:         fetch via current vector_A or vector_B
See aSOP:         store via current vector_A
See aVOAV:        validate vector_A:IWDD, vector_B:{IWDD,opnd},
                           length, stride_A, stride_B

Inv Arg/Par:      vector_A is sp and vector_B is dp
Vect Arith Exc:   R(B(i)*A(i)) = TooBig              [Exponent Overflow]
                  R(B(i)*A(i)) = TooSmall           [Exponent Underflow]
                  R(B(i)*A(i)) < > R*(B(i)*A(i))       [Precision Loss]
                  N(R(B(i)*A(i))) = TooSmall
                      but R = R* < > 0                        [spurious]
```

VUV <vector under (divided into) vector>

```
v   95EA

See aAPIX:        current vector_A by stride_A
                  current vector_B by stride_B
See aFOP:         fetch via current vector_A or vector_B
See aSOP:         store via current vector_A
See aVOAV:        validate vector_A:IWDD, vector_B:{IWDD,opnd},
                           length, stride_A, stride_B

Inv Arg/Par:      vector_A is sp and vector_B is dp
Vect Arith Exc:   R(B(i)/A(i)) = TooBig              [Exponent Overflow]
                  R(B(i)/A(i)) = TooSmall           [Exponent Underflow]
                  R(B(i)/A(i)) < > R*(B(i)/A(i))       [Precision Loss]
                  N(R(B(i)/A(i))) = TooSmall
                      but R = R* < > 0                        [spurious]
                  A(i) is 0                           [Divide by Zero]
```

WASD <write ASD-table word> (restricted)

v 95B3 See RASD

WAST <WASD trapping synonym> (restricted)

v 95A6 See RASD

WATI <read machine indentification> (restricted)

v 95A4 No interrupts

WHOI <read proc_id> (restricted)

v 954E No interrupts

WTOD <write time of day clock> (restricted)

v 9549

Unimplemented Op: (unconditional)

XFNC <external function call> (restricted)

v 959E See aMVST

XTND <set to double precision>

p CE

Inv Arg/Par: TOS not {operand, WordDD}
IWDD reindex disabled

Uninit Arg: TOS tag in {6, 4}

ZERO <insert literal zero>

p BO No interrupts

ZIC <zero Interrupt_Count> (restricted)

v 9540 No interrupts

Appendix H
Interrupt Reference

This appendix lists each interrupt in alphabetical order of the interrupt name. The condition group (CVI or OVI) is shown for each interrupt. The offset is shown for each OVI. The characterization as operator-dependent (ODI) or spontaneous (SI) is also shown.

The common actions and operators that invoke each ODI are listed alphabetically along with an indication of the reason for the invocation. The same notation and conventions are used as in Appendix G.

Absent Code Segment CVI 4 (04) ODI

The interrupt condition is ASD1.present_to_proc = 0 in the ASD designated by the object indicated.

```
aPRCW:          PCW or RCW
TEED & TEEU:    edtab
```

Absent Data Segment CVI 3 (03) ODI

The interrupt condition is ASD1.present_to_proc = 0 in the ASD designated by the object indicated.

```
aAPIX:          client paged unindexed DD (page table)
                ASD3 for client paged indexed DD (page table)
aEIXD:          client indexed DD
                ASD3 for client paged indexed DD (page table)
aEVLK:          if pseudolink = 0:
                ASD1[stack_num].present_to_proc = 0
                if pseudolink = 1:
                ASD1[seg_num].present_to_proc = 0
```

Activate Object CVI 12 (0C) ODI

```
LPOW:           reference ⟶▸ operand
```

Arithmetic Overflow

OVI 0 ODI

The arithmetic exception condition reported to software (defined in Section 4) is named in brackets following the arithmetic situation that causes the interrupt.

```
ADD:              R(x+y) = TooBig                    [Exponent Overflow]
DIVD:             R(x/y) = TooBig                    [Exponent Overflow]
                  divisor (TOS) is 0                    [Divide by Zero]
IDIV & RDIV:      divisor (TOS) is 0                    [Divide by Zero]
MULT & MULX:      R(x*y) = TooBig                    [Exponent Overflow]
SNGL:             RS(ND(x)) = TooBig                 [Exponent Overflow]
SNGT:             TS(ND(x)) = TooBig                 [Exponent Overflow]
SUBT:             R(x-y) = TooBig                    [Exponent Overflow]
```

Arithmetic Underflow

OVI 1 ODI

The arithmetic exception condition reported to software (defined in Section 4) is named in brackets following the arithmetic situation that causes the interrupt. The processor generates interrupts for a subset of those situations labeled "spurious".

```
ADD:              N(R(x+y)) = TooSmall                          [spurious]
DIVD:             R(x/y)  =  TooSmall                  [Exponent Underflow]
                  R(x/y) < > R*(x/y)                      [Precision Loss]
                  N(R(x/y)) = TooSmall but R = R* < > 0          [spurious]
MULT & MULX:      R(x*y)  =  TooSmall                  [Exponent Underflow]
                  R(x*y) < > R*(x*y)                      [Precision Loss]
                  N(R(x*y)) = TooSmall but R = R* < > 0          [spurious]
NORM:             N(x) = TooSmall                      [Exponent Underflow]
RDIV:             N(x MOD y) = TooSmall                         [spurious]
SNGL:             N(x) = TooSmall                      [Exponent Underflow]
                  RS(ND(x)) = TooSmall                 [Exponent Underflow]
                  NS(RS(ND(x))) = TooSmall             [Exponent Underflow]
SNGT:             N(x) = TooSmall                      [Exponent Underflow]
                  TS(ND(x)) = TooSmall                 [Exponent Underflow]
                  NS(TS(ND(x))) = TooSmall             [Exponent Underflow]
SUBT:             N(R(x-y)) = TooSmall                          [spurious]
```

Block Exit

CVI 11 (0B) ODI

```
EXIT etc.:        MSCW.block_exit=1
```

Bounds Error CVI 19 (13) ODI

```
aAPIX:          The following conditions detected by aAPIX cause
                client to generate interrupt if resulting
                (unrepresentable) DD is used or returned:
                     new virtual index < 0
                     unpaged DD and new index >= 2**20
                     unpaged CharDD and new word index >= 2**16
                     new page index > ActSegLen(PageTable)
                     new page index = ActSegLen(PageTable)
                        and new word index + new char index > 0
aEIRW:          displacement+offset >= ActSegLen
aEIXD:          word index >= ActSegLen
                 (except when ultimate client can generate
                  Source or Destination Boundary interrupt)
                page index >= ActSegLen (Page Table)
aEVLK:          displacement >= ActSegLen
aPOAV:          for client source or destination pointer
                   (blocked by zero-length operator that deletes
                    its arguments)
                  IWDD.index > ActSegLen
                  ICDD.word_index > ActSegLen
                  ICDD.word_index = ActSegLen and .char_index > 0
                  IWDD.index >= 2**16
                     and client is not Transfer Words
aPRCW:          (PCW or RCW).pwi >= ActSegLen
                (PCW or RCW).psi > 5
aVOAV:          MaxInx + 1 > |stride| > MaxInx/3
BRUN etc.:      op_pwi >= ActSegLen(PSN)
                op_psi > 5
DBUN etc.:      (branch-dest opnd.dyn_pwi)mod 2**14 >=
                  ActSegLen(PSN)
INDX:           RaI(index argument) > MaxInx
                new word index > ActSegLen
                new word index = ActSegLen and new char index > 0
NXLN, NXLV:     |RAI(index argument)| > MaxInx
OCRX:           RaI(index argument) not {1 to ICW.ICW_limit}
SE8C & SE4C:    IWDD.index >= 2**16 (implied by aAPIX)
                unpaged pointer word index > ActSegLen
SFDC, SFSC:     new word index > ActSegLen
                new word index = ActSegLen and
                   new char index > 0
                (Bounds error not generated by EXSD SFDC or EXSD SFSC)
TEED & TEEU:    edtab.ewi >= ActSegLen(table)
                edtab.esi > 5
TWSU:           updated source or destination ICDD has word_index
                   >= 2**16
```

A Bounds Error interrupt is also generated when code execution runs off the end of the code or edit-table segment. The latter case is reported as an ODI on the enter-table-edit operator and the former case might appear as an ODI on VARI or on the first syllable of a polysyllabic primary operator, but in general the situation cannot be related to a particular operator. Note also that the error will be reported only when the code stream execution crosses the segment boundary, not for any anticipatory code fetch.

Destination Boundary CVI 8 (08) ODI

The Destination Boundary condition is that the destination pointer designates the last+1 element of the destination sequence. For the relevent operators, this interrupt takes precedence over a Bounds Error interrupt (via aEIXD) on the destination pointer.

```
INSU etc., MCHR etc., TEQD etc., TRNS,
TUND & TUNU, TWFD etc., TWSD & TWSU,
UPUD etc.:          dest segment exhausted
```

Environmental Fault CVI 38 (26) SI

False Assertion OVI 0 ODI

```
AVER & ASRT:        Boolean (TOS) is False
```

Hardware Error - Here CVI 42 (2A) SI

Hardware Error - Lost CVI 37 (25) SI

Integer Overflow OVI 2 ODI

```
BCD,DBCD:           RId(B) = TooBig
DSRF etc.:          RId(argument to be scaled) = TooBig
DSRR:               RId(argument to be scaled) = TooBig
                       (when scale factor = 0)
                    TId(argument to be scaled) = TooBig
                       (when scale factor > 0)
DSRT:               TId(argument to be scaled) = TooBig
IDIV & RDIV:        x and y are sp and |x DIV y| >= 2**39
                    x or y is dp and |x DIV y| >= 2**78
NTGD:               RId(x) = TooBig
NTGR:               RaI(x) = TooBig
```

```
NTIA:              TI(x) = TooBig
NTTD:              TId(x) = TooBig
SCRF etc.:         RId(argument to be scaled) = TooBig
                      (when scale factor = 0)
                   TId(argument to be scaled) = TooBig
                      (when scale factor > 0)
SCRT:              TId(argument to be scaled) = TooBig
```

Invalid Access CVI 21 (15) ODI

```
aILRE:             read_only I1DD
aSOP:              read_only IWDD
INSU etc.,
MCHR etc.:         read_only destination pointer
OVRD etc.          read_only I1DD
TEQD etc., TRNS, TUND & TUNU, TWFD etc., TWOD, TWSD & TWSU,
UPUD etc.:         read_only destination pointer
```

Invalid Address - Here CVI 15 (0F) SI

Invalid Address - Lost CVI 32 (20) SI

Invalid Argument or Parameter CVI 17 (11) ODI

Checks on code parameters are so marked; all other checks are on stack arguments.

If the criterion for an Uninitialized Argument interrupt holds, that interrupt takes precedence over this one.

```
aEAC:              lambda > LL
aILRE:             interlock reference not {SIRW, I1DD, ASRW}
aPOAV:             client length not operand or |RaI(length)| > limit
                   client argument not specified type
                   reindex disabled in client indexed DD
                      (source, dest, set, trtab, or edtab)
                   EBCDIC ICDD.char_index > 5
                   hex    ICDD.char_index > 11
aSOP:              object is sp and reference is  I2DD
                   object is sp and SIRW ⟶▸ dp opnd
                   object is dp and reference is I1DD
                   object is dp and SIRW ⟶▸ {sp opnd, tag-6 word,
                                             tag-4 word}
```

aVOAV:	client length not sp integer or |length| > MaxInx client stride not sp integer or |stride| > MaxInx (client stride) * (restart count) > MaxInx client argument not specified type client vector IWDD reindex disabled
ADD, AMAX & AMIN:	x (TOS2) not operand or y (TOS) not operand
BCD:	B (TOS) not operand N > 24 (parameter check)
BMS:	length (TOS) not operand or |RaI(length)| > MaxInx domain (TOS2) not I1DD domain reindex disabled
CBON:	TOS not operand
CEQD etc.:	source1 is EBCDIC and source2 is hex or vice versa
CHSN:	TOS not operand
CPV etc.:	vector_A is sp and vector_B is dp or vice versa
DBCD:	N(TOS) not operand or RaI(N) not {0 to 24} B(TOS2) not operand
DBRS & DBST:	Db (TOS) not operand or RaI(Db) not {0 to 47}
DBUN etc.:	branch-dest (TOS) not {operand, PCW, SIRW}
DFTR:	Len (TOS) not operand or RaI(Len) not {0 to 48} Sb (TOS2) not operand or RaI(Sb) not {0 to 47} Db (TOS3) not operand or RaI(Db) not {0 to 47}
DINS:	Len (TOS2) not operand or RaI(Len) not {0 to 48} Db (TOS3) not operand or RaI(Db) not {0 to 47}
DISO:	Len (TOS) not operand or RaI(Len) not {0 to 48} Sb (TOS2) not operand or RaI(Sb) not {0 to 47}
DIVD:	x (TOS2) not operand or y (TOS) not operand
DRNT:	X (TOS3) or L (TOS2) or H (TOS) not operand
DSRF etc., DSRR, DSRT:	sf (TOS) not operand or RaI(sf) not {0 to 12} argument to be scaled (TOS2) not operand
EQUL etc.:	TOS not operand or TOS2 not operand
EVAL:	reference (TOS) not {SIRW, IWDD}
EXSD & EXSU:	source is EBCDIC and dest is hex or vice versa
FMN etc.:	vector_A is sp and V is dp or vice versa
GATH:	vector_A is sp and vector_B is dp or vice versa
GESZ:	TOS not touched DD
GINX:	TOS not pointer
GLEN:	TOS not {WordDD, CharDD}
IDIV & RDIV:	x (TOS2) not operand or y (TOS) not operand
INDX:	not (TOS in {WordDD, CharDD, SIRW} and TOS2 is operand or TOS is operand and TOS2 in {WordDD, CharDD, SIRW}) TOS or TOS2 is reindex-disabled IWDD or ICDD EBCDIC ICDD.char_index > 5 hex ICDD.char_index > 11

```
INSG & INOP:          dest is hex
JOIN:                 TOS is not operand or TOS2 not operand
LOAD:                 reference (TOS) not {SIRW, IWDD}
LODT:                 reference (TOS) not {SIRW, IWDD, ASRW,
                                           operand in {+0 to 2**29-1}}
LPOW:                 reference (TOS) not {SIRW, IWDD}
MULT & MULX:          x (TOS2) not operand or y (TOS) not operand
NORM:                 x (TOS) not operand
NORX:                 TOS not indexed DD
NOWR:                 TOS not touched DD
NTGD,
NTGR, NTIA, NTTD:     x (TOS) not operand
NXLN:                 not (      TOS in {SingleDD, SIRW} and
                                 TOS2 is operand
                           or TOS is operand and
                              TOS2 in {SingleDD, SIRW}    )
                      TOS or TOS2 is reindex-disabled I1DD
NXLV:                 not (      TOS in {WordDD, SIRW} and
                                 TOS2 is operand
                           or TOS is operand and
                              TOS2 in {WordDD, SIRW}    )
                      TOS or TOS2 is reindex-disabled IWDD
OCRX:                 ICW (TOS) not sp opnd or index (TOS2) not opnd
OVRD etc.:            reference (TOS) not {SIRW, I1DD, ASRW}
POLY:                 vector_A or vector_B is dp and Z is sp
RASD etc.:            specifier (TOS) not operand or |RaI(TOS)| > MaxInx
                      ASD_num (TOS2) not operand or |RaI(TOS2)| > MaxInx
RNGT:                 X (TOS) not operand
RPRR:                 reg-id (TOS) not operand
                      RaI(reg-id) not in {0-15, 37-38, 52-53}
RSQR:                 x (TOS) not operand or x < 0
SCAT:                 vector_A is sp and vector_B is dp or vice versa
SCRF etc.,
SCRR, SCRT:           argument to be scaled (TOS) not operand
                      ScaleFactor > 12 (parameter check)
SEQ:                  vector_A is sp and vector_B is dp
                      vector_A is sp and Z is dp or vice versa
SESW & SEDW:          TOS not {WordDD, CharDD}
                      ICDD.char_index < > 0
                      pointer reindex disabled
SE8C & SE4C:          TOS not {WordDD, CharDD}
                      ICDD.char_index < > 0
                      pointer reindex disabled
SNGL:                 x (TOS) not operand
SNGT:                 x (TOS) not {operand, WordDD}
                      IWDD reindex disabled
```

```
SPLT:              TOS not operand
SPRR:              reg-val (TOS) not operand
                   reg-id (TOS2) not operand or |RaI(reg-id)| > MaxInx
                     or RaI(reg-id) in {0-15, 37, 53}
SRCH:              domain (TOS) not I1DD
                   domain reindex disabled
STAG:              tag value (TOS) not operand
STOD & STON:       not (    TOS in {SIRW, IWDD} and TOS2 is operand
                         or TOS is operand and TOS2 in {SIRW, IWDD}  )
SUBT:              x (TOS2) not operand or y (TOS) not operand
SUM & SUMA:        vector_A is sp and sum is dp
SXSN:              TOS not operand
TEED & TEEU:       source is EBCDIC and dest is hex or vice versa
TEQD etc., TUND & TUNU,
TWFD etc.:         source is EBCDIC and dest is hex or vice versa
VALC & LVLC:       restart TOS not {SIRW, IWDD, operand}
VMV, VMVN, VOV:    vector_A is sp and vector_B is dp
VPS:               vector_A is sp and vector_B is dp
                   vector_A is sp and S is dp or vice versa
VPV:               vector_A is sp and vector_B is dp
VPVS, VSPV, VTS:   vector_A is sp and vector_B is dp
                   vector_A is sp and S is dp or vice versa
VTV, VUV:          vector_A is sp and vector_B is dp
WTOD:              TOS is not 36-bit integer
XTND:              TOS not {operand, WordDD}
                   IWDD reindex disabled
```

Invalid Code Word — CVI 35 (23) SI

Invalid Operator — CVI 22 (16) ODI

```
NVLD:              (unconditional)
```

Invalid Target — CVI 14 (OE) ODI

```
aCHAR:             client dest ICDD ⟶▸ not operand
                   client source ICDD ⟶▸ not operand
aFOP:              2nd word via SIRW or address couple
                     not operand
                   I1DD ⟶▸ not operand   (except for LOAD)
                   I2DD ⟶▸ not operand
                   2nd word via I2DD not operand
aILRE:             reference ⟶▸ not {sp operand, POW}
aSOP:              IWDD ⟶▸ not Data Word
                   2nd word not Data Word
DBUN etc.:         SIRW ⟶▸ not PCW
GATH:              current vector_I ⟶▸ not single_integer
                   |vector_I single_integer| > MaxInx
```

INDX:	SIRW ⟶▸ not {SIRW, WordDD, CharDD, untouched DD}
	SIRW ⟶▸ reindex-disabled IWDD or ICDD
LOAD:	SIRW or I1DD ⟶▸ not {SIRW, DD, Data Word}
LPOW:	SIRW ⟶▸ not {SIRW, IWDD, PCW, operand, POW}
	IWDD ⟶▸ not {operand, POW}
MPCW:	(sdll,sdi) ⟶▸ not {CSD, untouched CSD}
NXLN:	SIRW ⟶▸ not {SIRW, SingleDD, untouched DD}
	SIRW ⟶▸ reindex-disabled I1DD
	SingleDD[index argument] ⟶▸ not {untouched DD, unindexed DD}
NXLV:	SIRW ⟶▸ not {SIRW, WordDD, untouched DD}
	SIRW ⟶▸ reindex-disabled IWDD
SCAT:	current vector_I ⟶▸ not single_integer
	|vector_I single_integer| > MaxInx
STOD & STON:	SIRW ⟶▸ not {SIRW, IWDD, PCW, untouched DD, Data Word}
SWFD etc.:	set [WordIndex(c)] ⟶▸ not operand
TRNS:	trtab[WordIndex(c)] ⟶▸ not operand
TWFD etc.:	set[WordIndex(c)] ⟶▸ not operand
TWSD & TWSU:	dest pointer ⟶▸ not Data Word
	source pointer ⟶▸ not operand
VALC & LVLC:	address couple or SIRW ⟶▸ not {SIRW, IWDD, PCW, untouched DD, opnd}

Locking OVI 0 ODI

LOK:	interlock status bit Free

Loop Timer - Here CVI 41 (29) SI

Loop Timer - Lost CVI 36 (24) SI

Memory Data Error CVI 16 (10) SI

Message Error CVI 43 (2B) SI

Missing PCW CVI 2 (02) ODI

aINTE:	SIRW ⟶▸ not {SIRW, PCW}
ENTR:	Stk[F+1] not {SIRW, PCW}
	SIRW ⟶▸ not {SIRW, PCW}

MSM Hung CVI 39 (27) SI

Pack Unindexed DD CVI 47 (2F) ODI

ICUD etc., PKUD etc.:	source is unpaged unindexed copy DD

Service Request CVI 18 (12) SI

Source Boundary CVI 9 (09) ODI

The Source Boundary condition is that the source pointer designates the last+1 element of the source sequence for a pointer operator that transfers data from a source to a destination. For the relevant operators, this interrupt takes precedence over a Bounds Error interrupt (via aEIXD) on the source pointer.

```
MCHR etc.,
TEQD etc.,
TRNS,
TUND & TUNU,
TWFD etc.,
TWSD & TWSU:        source segment exhausted
```

Stack Overflow CVI 33 (21) SI

Stack Overflow, while caused by operator action, is defined as a spontaneous interrupt. Any operator that produces more words of stack output than it consumes as stack arguments increases the size of the expression stack. The actual interrupt generation may occur because of an operator like PUSH or ENTR that has no net effect on the size of the stack, but forces expression-stack values to migrate from processor registers to the stack in memory.

Stack Underflow CVI 34 (22) SI

Stack Underflow interrupts can be generated by or on behalf of any operator that consumes one or more arguments from the top of the stack. These situations are not listed as operator-dependent interrupts; they are considered spontaneous interrupts generated by the stack-adjust mechanism.

Structure Error CVI 20 (14) ODI

```
aACCE:              Stk[F].history_link_I >= 2**13 (frame overflow)
aAPIX:              ASD3 for paged indexed DD is not unpaged indexedDD
                       (not checked if page index known)
                    ASD3 ⟶ not ASD1/ASD2 (page table)
                    paged unindexed DD ⟶ not ASD1/ASD2 (page table)
                    PageTable[new page index]
                       not {unpaged unindexed DD, untouched DD}
aEIXD:              client indexed DD ⟶ not ASD1/ASD2
                    ASD3 for paged indexed DD is not unpaged indexedDD
                       (not checked if page index known)
                    ASD3 ⟶ not ASD1/ASD2 (page table)
                    PageTable[page index] not {unpaged unindexed DD,
                        untouched DD}
```

aEVCH:	for each AR traversed on environmental chain: MSCW.env_link or D register ⟶▸ not entered MSCW

aEVLK:	if pseudolink = 0:
	stack_num ⟶ not ASD1/ASD2
	if pseudolink = 1:
	seg_num ⟶ not ASD1/ASD2

aINTE:	Stk[F].history_link_I >= 2**13 (frame overflow)
aMVST:	Stack_number ⟶ not ASD1/ASD2 destination Stk[0] not TSCW
aPOAV:	ASD3 for client paged indexed DD is not unpaged indexed DD (not checked if page index known)
aPRCW:	PCW or RCW ⟶ not ASD1/ASD2
aVOAV:	ASD3 for client paged indexed DD is not unpaged indexed DD (not checked if page index known)
ENTR:	Stk[F] not inactive MSCW Stk[F].history_link_I >= 2**13 (frame overflow)
EXIT etc.:	Stk[D[LL]+ 1] not RCW Stk[D[LL]] not entered MSCW with i initially = D[LL], while (i = D[LL]) or (Stk[i] not entered MSCW): Stk[i] not MSCW (inactive or entered) or MSCW.history_link_I >= 2**13 (frame overflow) i := i - (history link)
GINX:	ASD3 for paged indexed DD is not unpaged indexedDD (not checked if page index known)
GLEN:	unpaged or unindexed DD ⟶ not ASD2 ASD3 for paged indexed DD is not unpaged indexedDD (not checked if page index known) paged indexed DD PageTable.ASD_ref ⟶ not ASD2
INDX:	DD ⟶ not ASD1/ASD2
SEDW & SESW:	ASD3 for paged indexed DD is not unpaged indexedDD (not checked if page index known)
SE4C & SE8C:	unpaged pointer ⟶ not ASD1/ASD2
SFDC,	
SFSC, SRDC, SRSC:	ICDD ⟶ not ASD1/ASD2
TEED & TEEU:	edtab ⟶ not ASD1/ASD2

Touch CSD

CVI 1 (01) ODI

MPCW:	(sdll,sdi) ⟶▸ untouched CSD

Touch Load

CVI 6 (06) ODI

LOAD:	SIRW or I1DD ⟶▸ untouched DD
NXLN:	SingleDD[index argument] ⟶▸ untouched DD

Touch Reference

CVI 5 (05) ODI

aAPIX:	PageTable[new page index] is untouched DD
aEIXD:	PageTable[page index] is untouched DD
INDX, NXLN, NXLV, STOD & STON:	SIRW ⟶▸ untouched DD
VALC & LVLC:	parameter or SIRW ⟶▸ untouched DD

Undefined Operator

CVI 23 (17) ODI

This interrupt is generated for any opcode that is not defined in the current context.

It is also generated in the following circumstances:

EXPU:	next operator is ENDE, ENDF, or INSC next operator is MCHR, MVNU, MINS, MFLT, SFSC, or SRSC (which requires source)
EXSD & EXSU:	next operator is ENDE, ENDF, or INSC

Unimplemented Operator

OVI 3 ODI

The following operators are unimplemented by the A16 processor (but are emulated by software):

SCLF	OVI	260	(104)
DSLF	OVI	264	(108)
SCRT	OVI	268	(10C)
DSRT	OVI	272	(110)
SCRR	OVI	292	(124)
DSRR	OVI	296	(128)
ICVU	OVI	304	(130)
PACU	OVI	360	(168)
WTOD	OVI	808	(328)
CSCP	OVI	1152	(480)
SHOW	OVI	1404	(57C)

Uninitialized Argument

CVI 24 (18) ODI

aILRE:	interlock-reference tag in {6, 4}
aPOAV, aVOAV:	client argument tag in {6, 4}
ADD, AMAX & AMIN:	TOS or TOS2 tag in {6, 4}
BCD:	TOS tag in {6,4}
BMS:	length or domain tag in {6,4}
CBON:	TOS tag in {6, 4}

```
CHSN:                TOS tag in {6, 4}
DBCD:                TOS or TOS2 tag in {6, 4}
DBRS & DBST:         Db tag in {6,4}
DBUN etc.:           branch-dest (TOS) tag in {6,4}
DFTR:                Len or Sb or Db tag in {6,4}
DINS:                Len or Db tag in {6,4}
DISO:                Len or Sb tag in {6,4}
DIVD:                TOS or TOS2 tag in {6,4}
DRNT:                X or L or H tag in {6,4}
DSRF etc., DSRR,
DSRT, EQUL etc.:     TOS or TOS2 tag in {6,4}
EVAL,
GESZ, GINX, GLEN:    TOS tag in {6,4}
IDIV & RDIV,
INDX, JOIN:          TOS or TOS2 tag in {6,4}
LOAD, LODT, LPOW:    TOS tag in {6,4}
MULT & MULX:         TOS or TOS2 tag in {6,4}
NORM, NORX, NOWR, NTGD,
NTGR, NTIA, NTTD:    TOS tag in {6,4}
NXLN, NXLV, ORCX:    TOS or TOS2 tag in {6,4}
OVRD etc.:           reference tag in {6,4}
RASD etc.:           TOS or TOS2 tag in {6,4}
RNGT:                TOS tag in {6,4}
RPRR:                TOS tag in {6,4}
RSQR, SCRF etc., SCRR, SCRT, SESW & SEDW,
SE8C & SE4C:         TOS tag in {6,4}
SNGL, SNGT, SPLT:    TOS tag in {6,4}
SPRR:                TOS or TOS2 tag in {6,4}
SRCH:                domain tag in {6,4}
STOD & STON,
SUBT:                TOS or TOS2 tag in {6,4}
SXSN:                TOS tag in {6,4}
WTOD:                TOS or TOS2 tag in {6,4}
XTND:                TOS tag in {6,4}
```

Unlocking

OVI 0 ODI

```
UNLK:                interlock status not Lock_Uncontended
```

Vector Arithmetic Exception

OVI 0 ODI

The following operators generate the interrupt because of arithmetic exceptions, as noted. Only a subset of those cases labeled "spurious" generate the interrupt.

CPVS:	RS(ND(B(i))) = TooBig	[Exponent Overflow]
	RS(ND(B(i))) = TooSmall	[Exponent Underflow]
	NS(RS(ND(B(i)))) = TooSmall	[Exponent Underflow]
DOT, DOTX:	R(A(i)*B(i)) = TooBig	[Exponent Overflow]
	R(A(i)*B(i)) = TooSmall	[Exponent Underflow]
	R(A(i)*B(i)) < > R*(A(i)*B(i))	[Precision Loss]
	N(R(A(i)*B(i))) = TooSmall but R = R* < > 0	[spurious]
	R(sum+...) = TooBig	[Exponent Overflow]
	N(R(sum+...)) = TooSmall	[spurious]
POLY:	R(B(i)*Z) = TooBig	[Exponent Overflow]
	R(B(i)*Z) = TooSmall	[Exponent Underflow]
	R(B(i)*Z) < > R*(B(i)*Z)	[Precision Loss]
	N(R(B(i)*Z)) = TooSmall but R = R* < > 0	[spurious]
	R(A(i)+...) = TooBig	[Exponent Overflow]
	N(R(A(i)+...)) = TooSmall	[spurious]
SEQ:	R(Z + B(i)) = TooBig	[Exponent Overflow]
	N(R(Z + B(i))) = TooSmall	[spurious]
SUM & SUMA:	R(sum + A(i)) = TooBig	[Exponent Overflow]
	N(R(sum + A(i))) = TooSmall	[spurious]
VMV:	R(A(i)-B(i)) = TooBig	[Exponent Overflow]
	N(R(A(i)-B(i))) = TooSmall	[spurious]
VMVN:	R(B(i)-A(i)) = TooBig	[Exponent Overflow]
	N(R(B(i)-A(i))) = TooSmall	[spurious]
VOV:	R(A(i)/B(i)) = TooBig	[Exponent Overflow]
	R(A(i)/B(i)) = TooSmall	[Exponent Underflow]
	R(A(i)/B(i)) < > R*(A(i)/B(i))	[Precision Loss]
	N(R(A(i)/B(i))) = TooSmall but R = R* < > 0	[spurious]
	B(i) is 0	[Divide by Zero]

VPS:	R(B(i)+S) = TooBig	[Exponent Overflow]
	N(R(B(i)+(S)) = TooSmall	[spurious]
VPV:	R(A(i)+B(i)) = TooBig	[Exponent Overflow]
	N(R(A(i)+B(i))) = TooSmall	[spurious]
VPVS:	R(B(i)*S) = TooBig	[Exponent Overflow]
	R(B(i)*S) = TooSmall	[Exponent Underflow]
	R(B(i)*S) < > R*(B(i)*S)	[Precision Loss]
	N(R(B(i)*S)) = TooSmall but R = R* < > 0	[spurious]
	R(A(i)+...) = TooBig	[Exponent Overflow]
	N(R(A(i)+...)) = TooSmall	[spurious]
VSPV:	R(A(i)*S) = TooBig	[Exponent Overflow]
	R(A(i)*S) = TooSmall	[Exponent Underflow]
	R(A(i)*S) < > R*(A(i)*S)	[Precision Loss]
	N(R(A(i)*S)) = TooSmall but R = R* < > 0	[spurious]
	R(...+B(i)) = TooBig	[Exponent Overflow]
	N(R(...+B(i))) = TooSmall	[spurious]
VTS:	R(B(i)*S) = TooBig	[Exponent Overflow]
	R(B(i)*S) = TooSmall	[Exponent Underflow]
	R(B(i)*S) < > R*(B(i)*S)	[Precision Loss]
	N(R(B(i)*S)) = TooSmall but R = R* < > 0	[spurious]
VTV:	R(A(i)*B(i)) = TooBig	[Exponent Overflow]
	R(A(i)*B(i)) = TooSmall	[Exponent Underflow]
	R(A(i)*B(i)) < > R*(A(i)*B(i))	[Precision Loss]
	N(R(A(i)*B(i))) = TooSmall but R = R* < > 0	[spurious]
VUV:	R(B(i)/A(i)) = TooBig	[Exponent Overflow]
	R(B(i)/A(i)) = TooSmall	[Exponent Underflow]
	R(B(i)/A(i)) < > R*(B(i)/A(i))	[Precision Loss]
	N(R(B(i)/A(i))) = TooSmall but R = R* < > 0	[spurious]
	A(i) is 0	[Divide by Zero]

Index

A

aACCE <accidental entry>, 4-85, G-3
aAPIX <apply index to data descriptor>, 4-55, G-3
absent code segment, 3-11, H-1
absent data segment, 3-11, H-1
absent segment, 3-11
absolute address, description, 1-14
Absolute Store Reference Word (ASRW), 2-7
access checking, 4-52
access control, 1-9
  read only, 1-9
  reindex, 1-9
  write-access, 1-9
aCHAR <access word via ICDD>, G-5
activate object, 3-12, H-1
activation records
  description, 1-3
  environmental chain, 1-5
  historical chain, 1-5
  pseudo, 1-6
Actual Segment Descriptors (ASDs), 1-2
actual segments, description, 1-2
Actual Store Reference Word (ASRW), 1-7
ADD <add>, 4-22, A-1, G-15
address couple checks, 3-19
address couple evaluation, 1-6, 4-51
address couples, 1-4, 2-12
  CSD, 2-13
  fixed fence, 2-12
  variable fence, 2-12
addressing, 1-1, 1-10
  current environment, 1-6
  environment, 1-10
aEAC <evaluate address couple>, G-5
aEIRW <evaluate SIRW>, G-6
aEIXD <evaluate indexed DD>, G-6
aEVCH <traverse environmental chain>, G-7
aEVLK <evaluate environmental link>, G-7
aFOP <fetch operand value>, 4-63, G-8
aILRE <interlock reference evaluation>, G-9
aINTE <interrupt entry>, 3-1, G-9
alarm interrupts, 3-25 through 3-28
  hardware error, 3-25
  invalid address, 3-27
  loop timer, 3-27
  memory data error, 3-28
  message error, 3-28
  msm hung, 3-28
AMAX <arithmetic maximum>, 4-26, A-1, G-15
AMIN <arithmetic minimum>, 4-26, A-1, G-15
aMVST <move to stack>, G-10, E-1
  interrupt handling, E-2
aPOAV <pointer operator argument validation>, G-10
approximation functions, 4-20
aPRCW <distribute PCW/RCW code stream pointer>, G-12
argument value checks, 3-19
argument, description, 4-1
arithmetic exceptions, 4-11
arithmetic interrupts, 3-7 through 3-10
  arithmetic overflow, 3-8
  arithmetic underflow, 3-9
  exception conditions, 3-7
  integer overflow, 3-10
  spurious interrupts, 3-7

arithmetic operators, 4-21
ADD, 4-22
AMAX, 4-26
AMIN, 4-26
DIVD, 4-23
IDIV, 4-24
MULT, 4-23
MULX, 4-23
NORM, 4-26
RDIV, 4-25
SUBT, 4-22
arithmetic overflow, 3-8, H-2
divide by zero, 3-8
exponent overflow, 3-8
arithmetic underflow, 3-9, H-2
exponent underflow, 3-9
precision loss, 3-9
arrogate interlock, 4-78
ASD1, 2-6
ASD2, 2-6
ASD memory, 1-14
ASD Structure errors, 3-23
ASD table
description, 1-14
ASDs, 1-2
ASD table, 1-14
description, 2-5
fields, 2-6
memory, 1-14
structure errors, 3-23
aSOP <store operand value>, 4-70, G-13
ASRT <assert>, 4-88, A-1, G-15
ASRW, 1-7, 2-7
assertion operators, 4-88
ASRT, 4-88
AVER, 4-88
automatic page switching, 2-11
AVER <aver>, 4-88, A-1, G-15
aVOAV <vector-operator argument validation>, G-13

B

BCD <binary convert to decimal>, 4-37, A-1, G-15
binary to decimal conversion operators, 4-36
BCD, 4-37
DBCD, 4-37
binding request, 3-12
binding request token, 1-3
bit vector interpretation, 4-38 through 4-49
bit vector type transfer operators, 4-41
evaluate word structure operators, 4-43
literal operators, 4-40
logical operators, 4-38
word manipulation operators, 4-44 through 4-49
bit vector type transfer operators, 4-41
bit vector type transfer operators, 4-41
JOIN, 4-42
SPLT, 4-43
STAG, 4-41
XTND, 4-42
block exit, 3-12, H-2
BMS <bounded masked search for equal>, 4-94, A-1, G-16
Boolean accumulators, 1-13, B-1
Boolean operands, 2-4
bounds error, 3-16, H-3
branching operators, 4-79
BRFL, 4-80
BRTR, 4-80
BRUN, 4-80
DBFL, 4-81
DBTR, 4-81
DBUN, 4-81
dynamic branches, 4-80
static branches, 4-79
break interlock, 4-78
BRFL <branch false>, 4-80, A-1, G-16
BRST <bit reset>, 4-45, A-2, G-16
BRTR <branch true>, 4-80, A-2, G-16
BRUN <branch unconditional>, 4-80, A-2, G-16
BSET <bit set>, 4-45, A-2, G-16

C

calling sequences, 3-2
  POI, 3-2
  ROI, 3-2
CBON <count binary ones>, 4-44, A-2, G-16
CEQD <compare characters equal delete>, 4-104, A-2, G-17
CEQU <compare characters equal update>, 4-104, A-2, G-17
CGED <compare characters greater or equal delete>, 4-104, A-2, G-17
CGEU <compare characters greater or equal update>, 4-104, A-2, G-17
CGTD <compare characters greater delete>, 4-104, A-2, G-17
CGTU <compare characters greater update>, 4-104, A-2, G-17
chaining rules, 4-53
  notation, 4-53
character descriptor, 2-8
character insert operators, 4-120
  ENDF, 4-122
  INOP, 4-121
  INSC, 4-121
  INSG, 4-121
  INSU, 4-121
character move operators, 4-122
  MCHR, 4-122
  MFLT, 4-123
  MINS, 4-123
  MVNU, 4-122
character relational operators, 4-101
  scan operators, 4-101
    SEQD, 4-101
    SEQU, 4-102
    SGED, 4-101
    SGEU, 4-102
    SGTD, 4-101
    SGTU, 4-102
    SLED, 4-101
    SLEU, 4-102
    SLSD, 4-101
    SLSU, 4-102
    SNED, 4-101
    SNEU, 4-102
  transfer operators, 4-102
    TEQD, 4-102
    TEQU, 4-103
    TGED, 4-102
    TGEU, 4-103
    TGTD, 4-102
    TGTU, 4-103
    TLED, 4-102
    TLEU, 4-102
    TLSD, 4-102
    TLSU, 4-103
    TNED, 4-102
    TNEU, 4-103
character sequence compare operators, 4-103
  CEQD, 4-104
  CEQU, 4-104
  CGED, 4-104
  CGEU, 4-104
  CGTD, 4-104
  CGTU, 4-104
  CLED, 4-104
  CLEU, 4-104
  CLSD, 4-104
  CLSU, 4-104
  CNED, 4-104
  CNEU, 4-104
character sequence extraction operator, 4-109
  SISO, 4-109
character set membership operators, 4-105
  scan operators, 4-106
    SWFD, 4-106
    SWFU, 4-106
    SWTD, 4-106
    SWTU, 4-106
  transfer operators, 4-106
    TWFD, 4-107
    TWFU, 4-107
    TWTD, 4-107
    TWTU, 4-107
character skip operators, 4-119
  skip forward, 4-120
    SFDC, 4-120
    SFSC, 4-120
  skip reverse, 4-120
    SRDC, 4-120
    SRSC, 4-120

character translate operator, 4-107
  TRNS, 4-107
CharDD, 2-8
char_index, 2-9
CHEK <compute check-hash>, 4-134, A-2, G-17
CHSN <change sign>, 4-49, A-2, G-18
CLED <compare characters less or equal delete>, 4-104, A-2, G-18
CLEU <compare characters less or equal update>, 4-104, A-2, G-18
CLSD <compare characters less delete>, 4-104, A-3, G-18
CLSU <compare characters less update>, 4-104, A-3, G-18
CNED <compare characters not equal delete>, 4-104, A-3, G-18
CNEU <compare characters not equal update>, 4-104, A-3, G-18
code parameter value checks, 3-19
Code Segment Descriptors (CSDs), 1-1, 2-10
code segment dictionary, 1-5
code segments, 1-13
code stream, 1-13
  designation, 3-3
  pointer, 1-13
  distribution, 4-79
  resumption, 3-3
  defunct notation, 3-3
common actions,
  aACCE, 4-85, G-3
  aAPIX, 4-55, G-3
  aCHAR, G-5
  aEAC, G-5
  aEIRW, G-6
  aEIXD, G-6
  aEVCH, G-7
  aEVLK, G-7
  aFOP, 4-63, G-8
  aILRE, G-9
  aINTE, G-9
  alphabetical listing, G-1
  aMVST, E-1, G-10
  aPOAV, G-10
  aPRCW, G-12
  aSOP, 4-70, G-13
  aVOAV, G-13
  description, 4-2
computational operators, 4-8 through 4-49
  bit vector interpretation, 4-38 through 4-49
  linear index function operator, 4-49
  numeric operand interpretation, 4-8
Condition-Vectored Interrupts (CVIs), 3-2
consistency checks, 3-20
control structure access rights, 1-9
CPV <copy vector>, 4-128, A-3, G-19
CPVA <copy vector absolute>, 4-128, A-3, G-19
CPVD <copy vector double>, 4-128, A-3, G-19
CPVN <copy vector negated>, 4-128, A-3, G-19
CPVS <copy vector single>, 4-128, A-3, G-19
CSCP <communicate with system control processor>, E-3, A-3, G-20
CSD address couples, 2-13
CSDs, 1-1, 2-10
  untouched CSD, 2-10
CVIs, 3-2

D

data array operators, 4-92 through 4-135
  paged array operations, 4-92
  pointer operators, 4-95 through 4-125
    character relational operators, 4-101
    character sequence compare operators, 4-103
    character sequence extraction operator, 4-108
    character set membership operators, 4-105
    character translate operator 4-107
    decimal character sequence operators, 4-109
    edit operators, 4-115
    unconditional character transfer operators, 4-100
    word transfer operators, 4-114
  searching operators, 4-93
  vector operators, 4-125 through 4-135
    recurrence operators, 4-132
    reduction operators, 4-131
    search operators, 4-134
    vector assignment operators, 4-127
Data Segment Descriptors (DSDs), 1-1
data structure access, 2-20
  DDs, 2-21
  descriptors, 2-21
  operands, 2-20
  operators, 2-21
  PCWs, 2-21
  primitive code, 2-21
  SIRWs, 2-21

data type formats, D-1
ASD1, D-3
ASD2, D-3
ASRW, D-13
double precision/numeric, D-2
entered MSCW, D-9
entered MSCW with normal env_link, D-10
entered MSCW with pseudo env_link, D-10
environment link, D-7
hard POW, D-15
Inactive MSCW, D-9
IndexedCharDD, D-6
IndexedWordDD, D-6
interlock POW, D-14
PCW, D-12
PCW skeleton, D-11
RCW, D-12
restart POW, D-15
single or double precision/Boolean, D-2
single precision/ICW, D-1
single precision/numeric, D-1
SIRW, D-7
SIRW with normal env_link, D-8
SIRW with pseudo env_link, D-8
soft POW, D-14
tag-4 and tag-6 word, D-11
Touched CSD, D-4
TSCW, D-13
unindexed DD, D-5
untouched CSD, D-4
untouched DD, D-5
data word, 2-2
DBCD <dynamic binary convert to decimal>, 4-37, A-3, G-20
DBFL <dynamic branch false>, 4-81, A-3, G-20
DBRS <dynamic bit reset>, 4-46, A-3, G-20
DBST <dynamic bit set>, 4-46, A-3, G-20
DBTR <dynamic branch true>, 4-81, A-3, G-20
DBUN <dynamic branch unconditional>, 4-81, A-3, G-20
DDs, data structure access, 2-21
decimal character sequence operators, 4-109
input convert operators, 4-113
pack operators, 4-110
unpack operators, 4-111
unpack signed operators, 4-112
unpack unsigned operators, 4-112
decimal sequence operands, 2-4
descriptor attribute extraction operators, 4-62
GESZ, 4-63
GINX, 4-63
GLEN, 4-62
descriptor interpretation, 4-8
descriptors, 1-1
data structure access, 2-21
destination argument, 4-97
destination boundary, 3-18, H-4
lengthening destination segment, 3-18
DEXI <disable external interrupts>, 4-90, A-4, G-20
DFTR <dynamic field transfer>, 4-48, A-4, G-21
diagrams, reading, 4-1
digit sequence, 2-4
DINS <dynamic field insert>, 4-47, A-4, G-21
DISO <dynamic field isolate>, 4-47, A-4, G-21
display register, 1-6
DIVD <divide>, 4-23, A-4, G-21
Divide By Zero, 3-8, 4-11
DLET <delete top of stack>, 4-88, A-4, G-21
DOT <dot (inner) product single>, 4-132, A-4, G-22
DOTX <dot (inner) product extended>, 4-132, A-4, G-22
double precision floating point operand, 2-4
double precision operands, 2-2
DoubleDD, 2-8
double_integer, magnitude, 2-4
DRNT <dynamic range test>, 4-28, A-4, G-22
DSDs, 1-1
DSLF <dynamic scale left>, 4-32, A-4, G-22
DSRF <dynamic scale right final>, 4-36, A-4, G-23
DSRR <dynamic scale right rounded>, 4-35, A-4, G-23
DSRS <dynamic scale right save>, 4-34, A-4, G-23

DSRT <dynamic scale right truncate>, 4-34, A-4, G-23
DUPL <duplicate top of stack>, 4-89, A-4, G-23
dyadic operators, 4-21
dyadic vector operators
  CPV, 4-128
  CPVA, 4-128
  CPVD, 4-128
  CPVN, 4-128
  CPVS, 4-128
  VMV, 4-128
  VMVN, 4-128
  VOV, 4-128
  VPV, 4-128
  VTV, 4-128
  VUV, 4-128
dyadic vector operators with scalar, 4-129
  VPS, 4-129
  VPVS, 4-129
  VSPV, 4-129
  VTS, 4-129
dynamic branches, 4-80
  DBFL, 4-81
  DBTR, 4-81
  DBUN, 4-81
dynamic chain, 1-6

E

edit modes, 4-116, C-1
edit mode operators, 4-119, C-6
  character insert operators, 4-120
  character move operators, 4-122
  character skip operators, 4-119
edit op code, 4-2
edit operators, 4-115
  edit mode operators, 4-119
  character skip operators, 4-119
  enter-edit operators, 4-115
    enter-single-edit operators, 4-117
    enter-table-edit operators, 4-116
  miscellaneous edit operators, 4-124
    ENDE, 4-124
    RSTF, 4-124
EEXI <enable external interrupts>, 4-90, A-4, G-23
element_size conventions, 4-95
ENDE <end edit>, 4-124, A-4, G-23
ENDF <end float>, 4-122, A-5, G-23
enforcement interrupts, 3-16 through 3-24
  bounds error, 3-16
  destination boundary, 3-18
  false assertion, 3-18
  invalid access, 3-19
  invalid argument or parameter, 3-19
  invalid code word, 3-20
  invalid operator, 3-20
  invalid target, 3-21
  pack unindexed DD, 3-21
  source boundary, 3-21
  stack underflow, 3-22
  structure error, 3-22
    ASD, 3-23
    page, 3-24
    stack, 3-23
  undefined operator, 3-24
  unimplemented operator, 3-15
  uninitialized argument, 3-24
enforcing segment boundaries, 1-8
enter-edit operators, 4-115
  enter-single-edit operators, 4-117
    EXPU, 4-118
    EXSD, 4-118
    EXSU, 4-118
  enter-table-edit operators, 4-116
    TEED, 4-117
    TEEU, 4-117
  single edit mode, 4-116
  table edit mode, 4-116
enter-single-edit operators, 4-117
enter-table-edit operators, 4-116
entered MSCW, 2-17
  data formats, D-9 through D-10
ENTR <enter>, 4-83, A-5, G-24
environment link evaluation, 4-50
environment link
  description, 2-13
  elements, 1-5
  fields, 2-13
 normal, 2-13
  pseudo, 1-5
  pseudolink bit, 2-13

environmental chain, 1-5
environment link, 1-5
environmental fault, 3-29, H-4
EQUL <equal>, 4-27, A-5, G-24
EVAL <evaluate>, 4-68, A-5, G-24
evaluate word structure operators, 4-43
  CBON, 4-44
  LOG2, 4-44
  RTAG, 4-43
EXCH <exchange top of stack>, 4-89, A-5, G-24
EXIT <exit>, 4-86, A-5, G-24
exponent overflow, 3-8, 4-11
exponent underflow, 3-8, 4-11
expression stack control, 4-5
EXPU <execute single edit operator, single pointer update>, 4-118, A-5, G-25
EXSD <execute single edit operator delete>, 4-118, A-5, G-25
EXSU <execute single edit operator update>, 4-118, A-5, G-25
external interrupts, 3-28
  environmental fault, 3-28
  service request, 3-28
EXTF <external sign flip-flop>, B-2, B-3

F

false assertion, 3-18, H-4
field specifications, 2-1
fixed fence address couples, 2-12
FLTF <float flip-flop>, B-2, B-3
FLTR <field transfer>, 4-48, A-5, G-25
FMN <find minimum>, 4-135, A-5, G-25
FMX <find maximum>, 4-135, A-5, G-25
FMXA <find maximum absolute value>, 4-135, A-5, G-25

G

GATH <gather>, 4-130, A-5, G-26
GESZ <get element_size>, 4-63, A-5, G-26
GINX <get index>, 4-63, A-5, G-26
GLEN <get length>, 4-62, A-5, G-26
GREQ <greater than or equal to>, 4-27, A-5, G-26
GRTR <greater than>, 4-27, A-5, G-26

H

HALT <conditional processor halt>, A-6, E-4, G-27
hard POW, 2-19
hardware error, 3-25
hardware error - here, 3-25, H-4
hardware error - lost, 3-25, H-4
hidden state, program restrictions, 4-136, E-10
historical chain, 1-5
  history links, 1-5

I

I1DD, 2-9
I2DD, 2-9
ICDD, 2-9
ICLD <input convert left-signed delete>, 4-113, A-6, G-27
ICRD <input convert right-signed delete>, 4-113, A-6, G-27
ICUD <input convert unsigned delete>, 4-113, A-6, G-27
ICVD <input convert delete>, 4-113, A-6, G-27
ICVU <input convert update>, 4-114, A-6, G-27
IDIV <integer divide>, 4-24, A-6, G-27
IMKS <insert mark stack>, 4-83, A-6, G-27
inactive MSCW, 2-16
indexed copy DD, 1-7
Indexed Data Descriptor (IndexedDD), 2-9
IndexedCharDD, 2-9
  index field, 2-9
    char_index, 2-9
    word_index 2-9
IndexedDD, 2-9
IndexedDoubleDD, 2-9
IndexedSingleDD, 2-9
IndexedWordDD, 2-9
  index field, 2-9
IndexedWordDD evaluation, 4-52
  access checking, 4-52
INDX <index>, 4-57, A-6, G-28
INOP <insert overpunch>, 4-121, A-6, G-28
input convert operators, 4-113
  ICLD, 4-113
  ICRD, 4-113
  ICUD, 4-113
  ICVD, 4-113
  ICVU, 4-114

INSC <insert conditional>, 4-121, A-6, G-28
INSG <insert display sign>, 4-121, A-6, G-28
INSR <field insert>, 4-47, A-6, G-28
INSU <insert unconditional>, 4-121, A-6, G-29
integer magnitude checks, 3-20
integer overflow, 3-10, 4-11, H-4
integerize with rounding, 4-16
interlock operations, 4-74 through 4-78
  LOK, 4-76
  UNLK, 4-77
interlock POW, 2-19
interlocks, 4-74
  arrogate operation, 4-78
  break operation, 4-78
  contention, 4-76
  defined operations, 4-75
  locking, 4-74
  states, 4-75
  status function, 4-78
  unlocking, 4-74
interrupt conditions, 3-2
interrupt definition
  description, 3-6
 types, 3-6
interrupt format, 3-2
interrupt loop, 3-2
interrupt parameters, 3-4
interrupt slot, 3-2
interrupt vector, 3-2, 3-29 through 3-31
interrupts, 4-3, H-1
  absent code segment, H-1
  absent data segment, H-1
  activate object, H-1
  arithmetic overflow, H-2
  arithmetic underflow, H-2
  block exit, H-2
  bounds error, H-3
  description, 3-1
  destination boundary, H-4
  environmental fault, H-4
  false assertion, H-4
  hardware error - here, H-4
  hardware error - lost, H-4
  integer overflow, H-4
  invalid
    access, H-5
    address - here, H-5
    address - lost, H-5
    argument/parameter, H-5 through H8
    code word, H-8
    operator, H-8
    target, H-8
  locking, H-9
  loop timer - here, H-9
  loop timer - lost, H-9
  memory data error, H-9
  message error, H-9
  missing PCW, H-10
  msm hung, H-10
  operator-dependent ODI, 3-1
  pack unindexed DD, H-10
  service request, H-10
  source boundary, H-10
  spontaneous, 3-1
  stack overflow, H-10
  stack underflow, H-10
  structure error, H-11
  touch CSD, H-12
  touch load, H-12
  touch reference, H-12
  undefined operator, H-12
  unimplemented operator, H-13
  uninitialized argument, H-13
  unlocking, H-14
  vector arithmetic exception, H-14
Interrupt_Count register, 3-2
invalid
  access, 3-19, H-5
  address, 3-27
  address - here, 3-27, H-5
  address - lost, 3-27, H-5
  argument/parameter, 3-19, H-5 through H-8
  address couple checks, 3-19
  argument value checks, 3-19
  code parameter value checks, 3-19
  consistency checks, 3-20
  integer magnitude checks, 3-20
  reindex checks, 3-19
  type checks, 3-19
invalid code word, 3-20, H-8
invalid operator, 3-20, 4-136, H-8
invalid target, 3-21, H-8
invariants, operators, 4-3
INXA <index via address couple parameter>, A-6, F-1
IRW chain, 4-53
ISOL <field isolate>, 4-46, A-6, G-29
item, description, 1-1
IWDD, 2-9

J

JOIN <join two operands into double>, 4-42, A-6, G-29

L

LAND <logical and>, 4-39, A-7, G-29
length argument, 4-96
lengthening destination segment, 3-18
LEQV <logical equivalence>, 4-39, A-7, G-29
LESS, 4-27, A-7, G-29
lex level, description, 1-4
linear index function operator, OCRX, 4-49
linear sum operators, 4-131
  SUM, 4-131
  SUMA, 4-131
literal operators, 4-40
  LT8, 4-40
  LT16, 4-40
  LT48, 4-40
  ONE, 4-40
  ZERO, 4-40
LNMC <long name call>, 4-54, A-7, G-29
LNOT <logical not>, 4-39, A-7, G-29
LOAD <load>, 4-69, A-7, G-30
locking, 3-12, H-9
LODT <load transparent>, 4-69, A-7, G-30
LOG2 <leading one test>, 4-44, A-7, G-30
logical equality, 4-39
logical operators, 4-38
  LAND, 4-38
  LEQV, 4-39
  LNOT, 4-39
  LOR, 4-39
  SAME, 4-39
LOK <lock interlock>, 4-76, A-7, G-30
loop timer, 3-27
loop timer - here, 3-27, H-9
loop timer - lost, 3-27, H-9)
LOR <logical OR>, 4-39, A-7, G-30
LPOW <load POW>, A-7, E-4, G-30
LSEQ <less than or equal to>, 4-27, A-7, G-31
LT8 <insert 8 bit literal>, 4-40, A-7, G-31
LT16 <insert 16 bit literal>, 4-40, A-7, G-31
LT48 <insert 48 bit literal>, 4-40, A-7, G-31
LVLC <long value call>, 4-65, A-7, G-31

M

mantissa field, 2-3
Mark Stack Control Word (MSCW), 1-5
mathematical function operator, RSQR, 4-38
MaxInx, 2-11
MaxLen, 2-11
MCHR <move characters>, 4-122, A-7, G-31
memory data error, 3-28, H-9
memory word, 1-14
message error, 3-28, H-9
MFLT <move with float>, 4-123, A-8, G-31
MINS <move with insert>, 4-123, A-8, G-31
miscellaneous operators, 4-135
  NOOP, 4-135
  NVLD, 4-136
  PUSH, 4-136
  VARI, 4-136
missing PCW, 3-13, H-10
MKSN <mark stack bound to name call>, 4-82, A-8, G-31
MKST <mark stack>, 4-82, A-8, G-31
modified capability architecture, description, 1-8
MPCW <make PCW>, 4-59, A-8, G-32
MSCW, 1-5, 2-16
  block_exit bit, 2-17
  entered MSCW, 2-17
  history_link field, 2-17
  inactive MSCW, 2-16
MSM Hung, 3-28, H-10
MULT <multiply>, 4-23, A-8, G-32
multiprocessing, 1-4
multiprogramming, 1-4
MULX <extended multiply>, 4-23, A-8, G-32
MVNU <move numeric>, 4-122, A-8, G-32

N

NAMC <name call>, 4-54, A-8, G-32
NEQL, 4-27, A-8, G-32
nondigit, 4-109
NOOP <no operation>, 4-135, A-8, G-32
NORM <normalize>, 4-26, A-8, G-33
normal store operators, 4-70
  aSOP, 4-70
  STOD, 4-71
  STON, 4-71

normalization function N, 4-19
normalized values, 4-10
NORX <disable reindex>, 4-62, A-8, G-33
NOWR <disable write access>, 4-61, A-8, G-33
NTGD <integerize double precision rounded>, 4-30, A-9, G-33
NTGR <integerize rounded>, 4-28, A-9, G-33
NTIA <integerize truncated>, 4-29, A-9, G-34
NTTD <integerize double precision truncated>, 4-31, A-9, G-34
numeric interpretation operators, 4-21
  arithmetic operators, 4-21
  binary to decimal conversion operators, 4-36
  numeric relational operators, 4-27
  numeric type transfer operators, 4-28 through 4-31
  range test operators, 4-27
  scale left operators, 4-31
  scale right operators, 4-32 through 4-36
numeric operand interpretation, 4-8
  representable values, 4-9
numeric relational operators, 4-27
  EQUAL, 4-27
  GREQ, 4-27
  GRTR, 4-27
  LESS, 4-27
  LSEQ, 4-27
  NEQL, 4-27
numeric type transfer operators, 4-28 through 4-31
  NTGD, 4-30
  NTGR, 4-28
  NTIA, 4-29
  NTTD, 4-31
  SNGL, 4-29
  SNGT, 4-30
NVLD <invalid operator>, 4-136, A-9, G-34
NXLN <index and load name>, 4-67, A-9, G-34
NXLV <index and load value>, 4-66, A-9, G-35
NXVA <index and load value via address couple parameter>, A-9, F-1

O

object reference, forms, 1-7
obsolescent operators, 4-4
obsolete operators, 4-4
OCRX <occurs index>, 4-49, A-9, G-35
ODI, 3-1
  resumable, 3-4
OFFF <overflow flip-flop>, B-2, B-3
ONE <insert literal one>, 4-40, A-9, G-35
op code, 4-1
  edit, 4-2
  primary, 4-2
  variant, 4-2
operands, 2-2
  data structure access, 2-20
  double precision, 2-2
  real numeric, 2-3
  single precision, 2-2
operator encoding, C-1
  interpreting one-syllable op code, C-1
    edit mode, C-1
    primary modem, C-1
    variant mode, C-1
operator subdivision, 3-6
Operator-Dependent Interrupts (ODI), 3-1
Operator-Vectored Interrupts (OVI), 3-2
operators, 4-1
  applicability, 4-4
  classes, 4-4
    obsolescent, 4-4
    obsolete, 4-4
    partially restricted, 4-4
    restricted, 4-5
    unrestricted, 4-4
  computational, 4-8
  data structure access, 2-20
  order of execution, 4-2
  overwrite, 4-72
Overlapping Source and Destination, 4-98
overwrite operators, 4-72
  OVRD, 4-73
  OVRN, 4-73
  RDLK, 4-74
OVIs, 3-2
OVRD <overwrite delete>, 4-73, A-9, G-35
OVRN <overwrite non-delete>, 4-73, A-9, G-35

P

pack operators, 4-110, A-9, G-35
  PACU, 4-110
  PKCD, 4-110
  PKLD, 4-110
  PKRD, 4-110
  PKUD, 4-110
Pack Unindexed DD, 3-21, H-10
PACU <pack update>, 4-110, A-9, G-36
page boundary interrupt, 4-93
Page Structure errors, 3-22
page table, 2-10
  ASD_ref, 2-11
paged array operations, 4-92
paged virtual segments, 2-10
  MaxInx, 2-11
  MaxLen, 2-11
  page table, 2-10
parameter, description, 4-1
partially restricted operators, 4-4
PCW, 1-7
  code stream pointer, 2-15
  data structure access, 2-21
  description, 2-15
  evaluation, 4-52
  skeleton, 2-15
phased out operators, F-1
  INXA, F-1
  NXVA, F-1
  STAD, F-2
  STAN, F-2
PKCD <pack delete>, 4-110
PKLD <pack left-signed delete>, 4-110, A-9, G-36
PKRD <pack right-signed delete>, 4-110, A-9, G-36
PKUD <pack unsigned delete>, 4-110, A-9, G-36
POI, 3-2
  parameter value, 3-4
pointer, 4-95
pointer index restrictions, 4-124
pointer operators, 4-95 through 4-125
  character relational operators, 4-101
    scan operators, 4-101
    transfer operators, 4-102
  character sequence compare operators, 4-103
  character sequence extraction operator, 4-108
  character set membership operators, 4-105
  character translate operator, 4-107
  decimal character sequence operators, 4-109
    input convert operators, 4-113
    pack operators, 4-110
    unpack operators, 4-111
  delete variation, 4-99
  destination argument, 4-97
  edit operators, 4-115
    edit mode operators, 4-119
    enter-edit operators, 4-115
  element_size conventions, 4-95
  length argument, 4-96
  overlapping source and destination, 4-98
  pointer index restrictions, 4-124
  segment boundary, 4-96
  short source operators, 4-97
  source1 and source2 arguments, 4-98
  source argument, 4-97
  unconditional character transfer operators, 4-100
  update variation, 4-99
  word transfer operators, 4-114
POLY <polynomial recurrence>, 4-133, A-10, G-36
POW, forms, 2-18
precision loss, 3-8, 4-11
preconditions, operators, 4-3
primary mode, C-1
  operators, C-2
primary op code, 4-2
primitive code, data structure access, 2-21

procedure entry operators, 4-82
aACCE, 4-85
ENTR, 4-83
IMKS, 4-83
MKSN, 4-82
MKST, 4-82
procedure exit operators, 4-85
EXIT, 4-86
RETN, 4-87
RTN2, 4-87
procedure
description, 1-3
entry, 1-10
accidental, 1-11
explicit, 1-11
implicit, 1-11
interrupt 1-11
process stack, 2-16
process
creation, 1-4
description, 1-4
stack, 1-4
processor register, 1-6
processor state manipulation operators, 4-90
DEXI, 4-90
EEXI, 4-90
ROFF, 4-90
RPRR, 4-91
RSNR, 4-91
RTFF, 4-91
RTOD, 4-91
SXSN, 4-91
processor state operators, 4-79 through 4-92
assertion operators, 4-78
branching operators, 4-79
code stream pointer distribution, 4-79
processor state manipulation operators, 4-90
stack frame operators, 4-81
procedure entry operators, 4-82
procedure exit operators, 4-85
top-of-stack operators, 4-88
program code words, 1-13, 2-5
syllables, 1-13, 2-5
Program Control Words (PCW), 1-7, 2-15
program restrictions, 4-136, E-10
Program Segment Number (PSN), 1-13
Program Syllable Index (PSI), 1-13
Program Word Index (PWI), 1-13
program
description, 1-4
local objects, 1-4
Programmed Operator Interrupt POI, 3-2
Protected Object Word POW, 2-18
protection mechanisms, 1-8
pseudo activation records, 2-14
PSN, 1-13
PUSH <push working stack onto activation record>, 4-136, A-10, G-36

Q

quadratic sum operators, 4-132
DOT, 4-132
DOTX, 4-132

R

range test operators, 4-27
DRNT, 4-28
RNGT, 4-27
RASD <read ASD-table word>, A-10, E-4, G-37
RAST <RASD trapping synonym>, A-10, E-4, G-37
RCW, 1-6, 2-16
RDIV <remainder divide>, 4-25, A-10, G-37
RDLK <read and lock>, 4-74, A-10, G-37
read evaluation operators, 4-63 through 4-70
aFOP, 4-63
EVAL, 4-68
LOAD, 4-69
LODT, 4-69
LVLC, 4-65
NXLN, 4-67
NXLV, 4-66
VALC, 4-64

real numeric operands, 2-3
  mantissa field, 2-3
recurrence operators, 4-132
  CHEK, 4-134
  POLY, 4-133
  SEQ, 4-133
reduction operators, 4-131
  linear sum operators, 4-131
    SUM, 4-131
    SUMA, 4-131
  quadratic sum operators, 4-132
    DOT, 4-132
    DOTX, 4-132
reference chains, 4-52
  chaining rule notation, 4-53
  chaining rules, 4-53
  IRW chain, 4-53
  SIRW chain, 4-53
reference evaluation, 4-51
  address couple evaluation, 4-51
  IndexedWordDD evaluation, 4-52
  PCW evaluation, 4-52
  reference chains, 4-52
  SIRW evaluation, 4-51
reference generation and evaluation operators, 4-50 through 4-74
  reference generation operators, 4-54 through 4-60
    aAPIX, 4-55
    INDX, 4-57
    LNMC, 4-54
    MPCW, 4-59
    NAMC, 4-54
    STFF, 4-54
  reference modification operators, 4-60 through 4-62
    NORX, 4-62
    NOWR, 4-61
    SEsz 4-61
reference types, 2-12 through 2-15
  activation records, 2-13
  address couples, 2-12
  CSD address couples, 2-13
  environment links, 2-13
  fixed fence address couples, 2-12
  PCWs, 2-15
  SIRWs, 2-14
  variable fence address couples, 2-12
registers
  description, 1-12
  values, 1-12
reindex bit, 2-9
reindex checks, 3-19
representable values, 4-9
  normalized values, 4-10
restart configuration, 3-5
restart mode, 3-5
  description, 4-4
restart POW, 2-19
Restartable Operator Interrupt (ROI), 3-3
restricted operators, 4-5, E-1
  aMVST, E-1
  CSCP, E-3
  HALT, E-4
  LPOW, E-4
  RASD, E-4
  RAST, E-4
  REXI, E-5
  RUNI, E-5
  SHOW, E-5
  SPRR, E-5
  STOP, E-6
  TWOD, E-7
  WASD, E-7
  WAST, E-7
  WATI, E-8
  WHOI, E-8
  WTOD, E-8
  XNFC, E-9
  ZIC, E-10
restrictions due to hidden state, 4-136, E-10
resumption from interrupt, 4-3
RETN <return>, 4-87, A-10, G-37
Return Control Word (RCW), 1-6
REXI <return/exit from interrupt>, A-10, E-5, G-37
RNGT <range test>, 4-27, A-10, G-37
ROFF <read and reset overflow flip-flop>, 4-90, A-10, G-37
ROI, 3-2
  P2 parameter, 3-4
rounding functions, 4-12 through 4-16
  R, example, 4-12
  Rai, 4-16
  RaI, 4-16
  Ri, 4-15
  RI, 4-15
  RId, 4-15

RPRR <read processor register>, 4-91, A-10, G-38
RSDN <rotate stack down>, 4-90, A-10, G-38
RSNR <read SNR>, 4-91, A-10, G-38
RSQR <real square root>, 4-38, A-10, G-38
RSTF <reset flip-flop>, 4-124, A-10, G-38
RSUP <rotate stack up>, 4-89, A-10, G-38
RTAG <read tag>, 4-43, A-10, G-38
RTFF <read true-false flip-flip>, 4-91, A-10, G-38
RTN2 <return two items>, 4-87, A-10, G-38
RTOD <read time of day clock>, 4-91, A-11, G-38
RUNI <indicate running>, A-11, E-5, G-39

S

SAME <logical equality>, 4-39, A-11, G-39
scalars, 4-125
scale factor, 4-31
scale left operators, 4-31
  DSLF, 4-32
  SCLF, 4-32
scale right operators, 4-32 through 4-36
  DSRF, 4-35
  DSRR, 4-35
  DSRS, 4-34
  DSRT, 4-34
  SCRF, 4-35
  SCRR, 4-35
  SCRS, 4-33
  SCRT, 4-34
SCAT <scatter>, 4-130, A-11, G-39
scatter/gather operators, 4-130
  GATH, 4-130
  SCAT, 4-130
SCLF <scale left>, 4-32, A-11, G-39
SCRF <scale right final>, 4-35, A-11, G-39
SCRR <scale right rounded>, 4-35, A-11, G-39
SCRS <scale right save>, 4-33, A-11, G-40
SCRT <scale right truncate>, 4-34, A-11, G-40
SE4C <set element_size to hex characters>, 4-61, A-11, G-41
SE8C <set element_size to EBCDIC characters>, 4-61, A-11, G-41
search operators, 4-134
  FMN, 4-135
  FMX, 4-135
  FMXA, 4-135
searching operators, 4-93
  BMS, 4-94
  SRCH, 4-93
SEDW <set element_size to double words>, 4-60, A-11, G-40
Segment Boundary, 4-96
segments
  actual, 1-2
  code, 1-13
  description, 1-1, 1-14
  virtual, 1-2
SEQ <sequential recurrence>, 4-133, A-11, G-40
SEQD <scan while equal delete>, 4-101, A-11, G-40
SEQU <scan while equal update>, 4-102, A-11, G-40
service interrupts, 3-11 through 3-16
  absent code segment, 3-11
  absent data segment, 3-11
  absent segment, 3-11
  activate object, 3-12
  binding request, 3-12
  block exit, 3-12
  locking, 3-12
  missing PCW, 3-13
  stack overflow, 3-13
  touch, 3-14
  unlocking, 3-15
  vector arithmetic exception, 3-16
service request, 3-28, H-10
service requests, 4-3
SESW <set element_size to single words>, 4-60, A-11, G-41
SEsz <set element_size ... >, 4-60
  SE4C <set element_size to hex characters>, 4-61
  SE8C <set element_size to EBCDIC characters>, 4-61
  SEDW <set element_size to double words>, 4-60
  SESW <set element_size to single words>, 4-60
SFDC <skip forward destination characters>, 4-120, A-11, G-41
SFSC <skip forward source characters>, 4-120, A-12, G-42
SGED <scan while greater or equal delete>, 4-101, A-12, G-42

SGEU <scan while greater or equal update>, 4-102, A-12, G-42
SGTD <scan while greater delete>, 4-101, A-12, G-42
SGTU <scan while greater update>,4-102, A-12, G-42
short destination operators, 4-113
Short Source operators, 4-97
SHOW <primitive display>, A-12, E-5, G-42
SI, 3-1
  resumable, 3-4
single edit mode, 4-116
single precision floating point operand, 2-3
single precision operands, 2-2
SingleDD, 2-8
single_integer, magnitude, 2-3
SIRWs, 1-7
  chain, 4-53
  data structure access, 2-21
  description, 2-14
  evaluation, 4-51
SISO, 4-108, A-12, G-42
skip forward, 4-120
  SFDC, 4-120
  SFSC, 4-120
skip reverse, 4-120
  SRDC, 4-120
  SRSC, 4-120
SLED <scan while less or equal delete>, 4-101, A-12, G-42
SLEU <scan while less or equal update>, 4-101, A-12, G-42
SLSD <scan while less delete>, 4-101, A-12, G-43
SLSU <scan while less update>, 4-102, A-12, G-43
SNED <scan while not equal delete>, 4-101, A-12, G-43
SNEW <scan while not equal update>, 4-102, A-12, G-43
SNGL <set to single precision rounded>, 4-29, A-12, G-43
SNGT <set to single precision truncated>, 4-30, A-12, G-43
soft POW, 2-18
Source1 and Source2 Arguments, 4-98
source argument, 4-97
source boundary, 3-21, H-10
source, description, 4-44
SPLT <split operand into two singles>, 4-43, A-13, G-43
Spontaneous Interrupt (SI), 3-1
SPRR <set processor register>, A-13, E-5, G-44
spurious interrupts, 3-7
SRCH <masked search for equal>, 4-93, A-13, G-44
SRDC <skip reverse destination characters>, 4-120, A-13, G-44
SRSC <skip reverse source characters>, 4-120, A-13, G-44
stack frame linkage, 2-16
  MSCW, 2-16
  RCW, 2-16
  TSCW, 2-18
stack frame operators, 4-81
  procedure entry operators, 4-82
  procedure exit operators, 4-85
stack overflow, 3-13, H-10
stack references, 4-50
stack state transformation, 4-7
stack structure, 2-16
stack structure errors, 3-23
stack underflow, 3-22, H-10
stacks
  description, 1-4
  expression, current, 1-11
  frame, 1-4, 1-11, 1-12
STAD <store delete via address couple parameter>, A-13, F-2
STAG <set tag>, 4-41, A-13, G-44
STAN <store non-delete via address couple parameter>, A-13, F-2
static branches, 4-79
  BRFL, 4-80
  BRTR, 4-80
  BRUN, 4-80
static chain, 1-6
status interlock, 4-78
STFF <stuff>, 4-54, A-13, G-45
STOD <store delete>, 4-71, A-13, G-45
STON <store non-delete>, 4-71, A-13, G-45
STOP <unconditional processor halt>, A-13, E-6, G-45
strides, 4-126
structure error, 3-22, H-11
  ASD structure errors, 3-23
  page structure errors, 3-24
  stack structure errors, 3-23
  types 3-22

Stuffed Indirect Reference Word (SIRW), 1-7
SUBT <subtract>, 4-22, A-13, G-45
SUM <summation over vector>, 4-131, A-13, G-45
SUMA <summation over vector, absolute>, 4-131, A-14, G-46
superhalt, 3-1
  conditions, 3-1
  interrupt loop, 3-2
  Interrupt_Count register, 3-2
SWFD <scan while false delete>, 4-106, A-14, G-46
SWFU <scan while false update>, 4-106, A-14, G-46
SWTD <scan while true delete>, 4-106, A-14, G-46
SWTU <scan while true update>, 4-106, A-14, G-46
SXSN <set external flip-flop>, 4-91, A-14, G-46

T

table edit mode, 4-116
tag values, 2-19
  RTAG, 2-19
  STAG, 2-19
Tag-4 word, 2-5
Tag-6 word, 2-5
TEED <table enter edit delete>, 4-117, A-14, G-47
TEEU <table enter edit update>, 4-117, A-14, G-47
TEQD <transfer while equal delete>, 4-102, A-14, G-47
TEQU <transfer while equal update>, 4-103, A-14, G-47
TFFF <true-false flip-flop>, B-1, B-3
TGED <transfer while greater or equal delete>, 4-102, A-14, G-47
TGEU <transfer while greater or equal update, 4-103, A-14, G-47
TGTD <transfer while greater delete>, 4-102, A-14, G-47
TGTU <transfer while greater update>, 4-103, A-14, G-48
thunk, 1-11
TI function, 4-16
TId function, 4-16
Ti function, 4-16
TLED <transfer while less or equal delete>, 4-102, A-14, G-48
TLEU <transfer while less or equal update>, 4-103, A-15, G-48
TLSD <transfer while less delete>, 4-102, A-15, G-48
TLSU <transfer while less update>, 4-103, A-15, G-48
TNED <transfer while not equal delete>, 4-102, A-15, G-48
TNEU <transfer while not equal update>, 4-103, A-15, G-48
Top of Stack Control Word (TSCW), 2-16
top-of-stack operators, 4-88
  DLET, 4-88
  DUPL, 4-89
  EXCH, 4-89
  RSDN, 4-90
  RSUP, 4-89
top-of-stack pointer register, 4-6
top-of-stack pop operations, 4-6
top-of-stack push operations, 4-6
Touch, 3-14
  touch CSD, 3-14, H-12
  touch load, 3-14, H-12
  touch reference, 3-14, H-12
touched CSD, 2-8
TRNS <translate>, 4-107, A-15, G-48
truncate-to-integer, 4-18
truncation functions, 4-17
  T, 4-15
  Ti, 4-16
TI, 4-16
TId, 4-16
  TS, 4-16
TSCW, 2-16
TUND <transfer characters unconditional delete>, 4-100, A-15, G-49
TUNU <transfer characters unconditional update>, 4-100, A-15, G-49
TWFD <transfer while false delete>, 4-107, A-15, G-49
TWFU <transfer while false update>, 4-107, A-15, G-49
TWOD <transfer words overwrite delete>, A-15, E-7, G-49
TWSD <transfer words delete>, 4-115, A-15, G-50
TWSU <transfer words update>, 4-115, A-15, G-50
TWTD <transfer while true delete>, 4-107, A-15, G-50
TWTU <transfer while true update>, 4-107, A-15, G-50
type checks, 3-19

U

unconditional character transfer operators, 4-100
TUND, 4-100
TUNU, 4-100
undefined operator, 3-24, H-12
unimplemented operator, 3-15, H-13
operator emulation, 3-15
unindexed copy DD, 1-7
unindexed DD, 2-8
uninitialize single datum, 2-5
uninitialized argument, 3-24, H-13
UNLK <unlock interlock>, 4-77, A-15, G-50
unlocking, 3-15, H-14
unpack operators, 4-111
unpack signed operators, 4-112
UPLD, 4-112
UPLU, 4-112
UPRD, 4-112
UPRU, 4-112
USND, 4-112
USNU, 4-112
unpack unsigned operators, 4-112
UPUD, 4-112
UPUU, 4-112
unpaged virtual segment, 2-10
unrestricted operators, 4-4
untouched CSD, 2-10
untouched DD, 2-7
untouched original, 1-3
untouched VSD, 2-7
update, 4-99
UPLD <unpack left-signed delete>, 4-112, A-16, G-50
UPLU <unpack left-signed update>, 4-112, A-16, G-50
UPRD <unpack right-signed delete>, 4-112, A-16, G-51
UPRU <unpack right-signed update>, 4-112, A-16, G-51
UPUD <unpack unsigned delete>, 4-112, A-16, G-51
UPUU <unpack unsigned update>, 4-112, A-16, G-51
USND <unpack signed delete>, 4-112, A-16, G-51
USNU <unpack signed update>, 4-112, A-16, G-51

V

VALC <value call>, 4-64, A-16, G-51
VARI <introduce variant operator>, 4-136, A-16, G-51
variable fence address couples, 2-12
variant mode, C-1
variant mode operators, C-4
variant op code, 4-2
vector access, 4-126
vector arguments, 4-125
vector arithmetic, 4-126
Vector Arithmetic Exception, 3-16, H-14
vector assignment operators, 4-127
dyadic vector operators, 4-127
CPV, 4-128
CPVA, 4-128
CPVD, 4-128
CPVN, 4-128
CPVS, 4-128
VMV, 4-128
VMVN, 4-128
VOV, 4-128
VPV, 4-128
VTV, 4-128
VUV, 4-128
dyadic vector operators with scalar, 4-129
VPS, 4-129
VPVS, 4-129
VSPV, 4-129
VTS, 4-129
scatter/gather operators, 4-130
GATH, 4-130
SCAT, 4-130
vector length, 4-126
vector operators, 4-125 through 4-135
recurrence operators, 4-132
reduction operator, 4-131
scalars, 4-125
search operators, 4-134
strides, 4-126
vector access, 4-126
vector arguments, 4-125
vector arithmetic, 4-126
vector assignment operators, 4-127
vector length, 4-126
vectors, 4-125
vectors, 1-6, 4-125

virtual index
  components, 2-12
  description, 2-11
Virtual Segment Descriptors, (VSD), 1-2, 2-7
virtual segments, 2-10
  description, 1-2
  domain, 4-93
  paged, 2-10
    MaxInx, 2-11
    MaxLen, 2-11
    page table, 2-10
    unpaged, 2-10
VMV <vector minus vector>, 4-128, A-16, G-52
VMVN <vector minus vector negated>, 4-128, A-16, G-52
VOV <vector over (divided by) vector>, 4-128, A-16, G-52
VPS <vector plus scalar>, 4-129, A-16, G-53
VPV <vector plus vector>, 4-128, A-16, G-53
VPVS <vector plus vector times scalar>, 4-129, A-16, G-53
VSDs, 1-2, 2-7
  classes of, 1-2
  untouched, 1-3
VSPV <vector times scalar plus vector>, 4-129, A-16, G-54
VTS <vector times scalar>, 4-129, A-17, G-54
VTV <vector times vector>, 4-128, A-17, G-55
VUV <vector under (divided into) vector>, 4-128, A-17, G-55

W

WASD <write ASD-table word>, A-17, E-7, G-55
WAST <WASD trapping synonym>, A-17, E-7, G-55
WATI <read machine identification>, A-17, E-8, G-56
WHOI <read proc_id>, A-17, E-8, G-56
word descriptor, 2-8
word format, 2-1
  fields, 2-1
word manipulation operators, 4-44 through 4-49
  BRST, 4-45
  BSET, 4-45
  CHSN, 4-49
  DBRS, 4-46
  DBST, 4-46
  DFTR, 4-48
  DINS, 4-47
  DISO, 4-47
  FLTR, 4-48
  INSR, 4-47
  ISOL, 4-46
word transfer operators, 4-114
  TWSD, 4-115
  TWSU, 4-115
WordDD, 2-8
words
  description, 1-1, 2-1
  distinguishing types, 2-2
  format, 2-1
word_index, 2-9
WTOD <write time of day clock>, A-17, E-8, G-56

X

XFNC <system function call>, A-17, E-9, G-56
XTND <set to double precision>, 4-42, A-17, G-56

Z

ZERO <insert literal zero>, 4-40, A-17, G-56
ZIC <zero interrupt_count>, A-17, E-10, G-56

39508825-100